Programmable Architectures for the Automated Design of Digital FIR Filters using Evolvable Hardware

Benjamin Iain Hounsell

A thesis submitted for the degree of Doctor of Philosophy.
The University of Edinburgh.
September 2001

Abstract

Continuing increases in both the size and complexity of digital signal processing (DSP) systems places a considerable demand on the design engineer to develop hardware architectures capable of fulfilling the growing functional requirements expected of modern DSP devices. Automated circuit design techniques provide the design engineer with a tool to more effectively generate high performance signal processors capable of meeting demanding specifications.

Evolvable hardware (EHW) is a relatively new approach to automated circuit design which utilises advances in reconfigurable hardware technology and the power of modern micro processors to generate circuits based on the principles of natural selection and evolution. This thesis investigates the suitability of software-biased and hardware oriented programmable platforms, configured via EHW, and tailored for the automated design of high performance DSP circuits. Performance criteria such as timing, area and circuit robustness are considered.

A number of benchmarked DSP circuits were initially considered. It was shown that by using larger functional logic macros as building blocks EHW is more successful at generating circuit solutions than if only gate primitives are used. In addition, the circuits generated are of comparable or better performance than equivalent circuits developed using a standard digital design methodology. Results also indicated that for more complex DSP functions to be generated, EHW platforms must use larger functional blocks, constrained for a specific application.

Finite Impulse Response (FIR) filters were identified as the key function of many DSP applications, and the multiplication unit was targeted as the performance critical component. A novel Programmable Arithmetic Logic Unit (PALU) was therefore developed as a functional building block suitable for automated digital filter design using EHW. The PALU replaces coefficient multiplication with a series of bit-shifts, additions and subtractions. Two distinct arrays of PALU were developed based on conventional FPGA and PLA re-configurable hardware architectures. Results show that a PLA architecture with 2 levels of hierarchical interconnect and column-based fixed tap outputs provides a platform most suited to automated filter design using the EHW technique. The PLA was also shown to be be robust to faults covering up to 25% of the array when configured using EHW.

Declaration of originality

I hereby declare that the research recorded in this thesis and the thesis itself was composed and originated entirely by myself in the Department of Electronics and Electrical Engineering at The University of Edinburgh.

Benjamin Hounsell

Acknowledgements

Firstly, I would like to thank my Supervisor Dr Tughrul Arslan for his excellent support and guidance during the last 3 years.

Considerable thanks to Dr Alan Murray, Dr Alister Hamilton, Dr Gerard Alan and Dr Ahmet Erdogan for their worthwhile discussions and invaluable feedback throughout.

Grateful thanks to Applied Materials whose kindly provided the financial support for my thesis.

Finally, many thanks to all those in the Department of Electronics and Electrical Engineering at the University of Edinburgh who made the last 3 years both rewarding and enjoyable.

Contents

List of figures

List of tables

Acronyms and Abbreviations

AAOS	Alternating Arrow Output Sequence
AFFA	Alternating Feed-forward Array
ALU	Arithmetic Logic Unit
AOOS	Alternating Orthogonal Output Sequence
ASIC	Application Specific Integrated Circuit
BLIS	Base-line Input Sequence
BLOS	Base-line Output Sequence
CFFA	Continuous Feed-forward Array
CFFLA	Continuous Feed-forward Loop Array
CSD	Canonic Signed Digit
CLB	Configurable Logic Block
DA	Distributed Arithmetic
DSP	Digital Signal Processing
EA	Evolutionary Algorithm
EHW	Evolvable Hardware
EOS	Edged Output Sequence
EP	Evolutionary Programming
ES	Evolutionary Strategies
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
GP	Genetic Programming
HDL	Hardware Description Language
IC	Integrated Circuit
LSB	Least Significant Bit
LSIS	L-Shaped Input Sequence
LUT	Look-up-table
MAC	Multiply Accumulate

MSB	Most Significant Bit
MUX	Multiplexer
PALU	Programmable Arithmetic Logic Unit
PLA	Programmable Logic Array
PLD	Programmable Logic Device
POF	Primitive Operator Filter
RTL	Register Transfer Language
VHDL	Very High Speed Integrated Hardware Description Language

Nomenclature

$AAOS_{taps}$	Number of CLBs available as potential taps using AAOS
$AOOS_{taps}$	Number of CLBs available as potential taps using AOOS
$BLOS_{taps}$	Number of CLBs available as potential taps using BLOS
B_i	Output Bits
D_i	Boltzmann probability distribution
EOS_{taps}	Number of CLBs available as potential taps using EOS
F_i	Bit-wise fitness
F	Distributed arithmetic multiplication function
$H(\omega)$	FIR filter frequency response
$H_{A(\omega)}$	Actual amplitude response of FIR filter
$H_{D(\omega)}$	Desired amplitude response of FIR filter
$H_{a(\omega)}$	Convolution of $H_{A(\omega)}$ with $H_{D(\omega)}$
$H(z)$	FIR filter transfer function
I	Total number of bits required to determine FPGA input of X(n)
λ	Offspring population
μ	Parent population
$PALU_{cntrl}$	Number of bits required to program 1 PALU
P_M	Bit-wise mutation probability
Q_x	Coefficient fitness score for evolvable hardware
R_c	Number of control bits required for routing MUX
S_c	Control bits of left-shift operator for first column of PALUs
S_{FPGA}	Total bit length of configuration string for FPGA-based filter
S_i	Search Space using Virtual Chip
$s(n)$	FIR filter sample response
S_{PLA}	Total bit length of configuration string for PLA-based filter
W	Bit length required to encode 1 PALU
$y(n)$	FIR filter difference equation
X_K	Input string description using distributed arithmetic

$X(n)$	FIR filter input
$Xwidth$	Number of PALU Columns
$Ywidth$	Number of PALU Rows

Chapter 1
Introduction

1.1 Automated Circuit Design

The continued development of more complex electronic devices with increasing integrated functionality translates into an increase in the size and complexity of the digital systems employed. This places a considerable demand on the design engineer to develop hardware architectures capable of fulfilling the growing functional requirements expected from these modern electronic devices, such as mobile phones and personal digital assistants (PDAs). Digital Signal processing (DSP) systems are extensively used in many electronic devices, performing tasks from signal filtering to data compression and real-time video streaming. Performance constraints such as operational speed, physical area, low power consumption, design portability and device reliability, all of which contribute directly to design complexity, are increasingly dominant factors when developing very large scale integrated (VLSI) silicon devices. This is especially important for highly constrained portable electronic device such as mobile phones. Automated circuit design techniques provide the design engineer with a tool to more effectively generate high performance electronic processors capable of meeting these demanding specifications. Circuit complexity, physical hardware constraints, and device flexibility and adaptation are all aspects of circuit design which can benefit from the design automation paradigm.

A number of automated design techniques for specific types of DSP circuit have been proposed and include the automated design of high speed multiplication-and-accumulation circuits [1] and VLSI digital FIR filters [2]. To obtain functionality an expert system, or heuristic knowledge of the system under design is often required, or design parameters and algorithms must be painstakingly developed which are often specific to the application. More recent developments in circuit design automation have resulted in the emergence of an independent and expanding field of research termed *evolvable hardware* (EHW) which is based on a non-heuristic search technique.

1.1.1 Evolvable Hardware

Digital circuit design automation using EHW differs from traditional IC design techniques which utilise a top-down or compartmentalised methodology in which complex systems are broken down and designed as smaller sub-systems. EHW instead approaches the design problem as a flat component hierarchy, generating a black-box of the completed circuit. This is achieved by using a number of circuit building blocks ranging from simple gate primitives to more complex digital signal processing elements. The size of building block relates to the level of design abstraction. When using gate primitives, the level of design abstraction can be likened to a bottom-up design approach, as many gates must be instantiated in order to achieved the desired circuit functionality. This requires considerable design effort. When more complex macro functions are employed, the design problem becomes more like a top-down approach, because much of the functionality of the desired circuit can be described using fewer building blocks and less design effort. Often in EHW the design abstraction is a combination of the two. Designs are generated autonomously via a group of stochastic optimisation techniques termed *evolutionary algorithms* (EAs). Evolutionary algorithms have been shown to be valuable in applications such as neural networks [3–5], task scheduling [6], VLSI routing [7, 8] and networking within telecommunications systems [9]. These are tasks where the computational time needed to provide a solution grows exponentially with problem complexity, and is termed NP-complete. The automated design of a number of digital circuits has also been shown to be an NP-complete [10] problem. This has lead to the development of a number of signal processing applications which utilise EHW design techniques. These include the development of an adaptive control for a myoelectric hand [11], the generation of novel multiplier circuits [12], and the exploitation of the physical properties of silicon in the design of a highly area efficient tone discriminator [13]. The flexibility and wide applicability of EHW for a range of automated design applications stems from the non-heuristic evolutionary algorithm it employs. The following three examples demonstrate how evolvable hardware can be applied to solve critical development issues associated with modern DSP circuit design.

Example 1: Ongoing advances in silicon manufacture have resulted in the widespread adoption of DSP hardware developed using deep sub-micron technologies. Commercially available applications utilising transistor sizes of 0.18 microns are now common place. System-on-chip (SoC) design methodologies have been developed to exploit the high transistor density inherent in deep sub-micron devices. SoC therefore facilitates the integration of many smaller data

processing tasks to form more complex signal processing applications, which often require tens of millions of transistors on a single chip. The time required to both design and test the complex functionality of SoC applications is regarded as the limiting factor in the products time-to-market. Circuits generated through EHW have been shown to reduce circuit area, and provide novel DSP architectures beyond those attainable through conventional design techniques [14, 15]. Circuit test is considered an integral part of the design automation procedure. However concerns over the complexity issues associated with rigorously evaluating and testing the functional correctness of circuits developed using EHW presents a number of problems for the automated design of complex DSP applications [16]. EHW has therefore been cited as a means of generating adaptive systems where the target function is not clearly understood such that test vectors can be added over an extended period whilst the device is in operation.

Example 2: The integration of communication media, such as telecommunication and real-time video images in mobile devices, require systems capable of rapid data manipulation and adaptation to both the changing requirements of the user, and the changing environment in which the device is deployed. Programmable, multi-purpose architectures are therefore required to provide a number of signal processing tasks on demand. Micro-processors and programmable DSP devices have traditionally been employed for such applications. However general purpose Programmable Logic Devices (PLD's) are now increasingly favoured as they are low cost and can be configured to produce user-defined architectures, specific to a signal processing task. PLD's allow the design engineer to down-load circuit configurations onto hardware an almost unlimited number of times. This provides the designer with a means of both implementing a signal processing device, and testing it, without the need to fabricate a full-custom IC. Applications using EHW have recently been introduced which exploit the programmability of PLD's by modifying device functionality in real time. For example, Tufte and Haddow have developed a programmable digital signal filter implemented on a PLD and capable of self-configuring for new tasks using EHW [17].

Example 3: Many DSP devices are deployed under hostile operating conditions for example in commercial satellite communications where hardware deteriorates due to damage caused by harmful radiation, and in other inhospitable environments where human intervention is difficult or impossible. These systems must therefore maintain functionality despite factors such as severe temperature variation, radiation, and operational ware. However, architectures developed using conventional fault tolerant design methodologies restrict DSP performance as

they reduce operational speed, and increase physical area [18]. EHW implementations of fault tolerant applications reduces the physical resources required, either by designing built-in robustness into the architecture as in [19, 20], or providing a single fixed-sized resource capable of adapting the system should it become damaged [21, 22]. In the latter case programmable logic devices must be employed.

1.1.2 Evaluating Circuits Through Evolvable Hardware

Circuits generated through EHW are evaluated either *extrinsically* through software simulation, or *intrinsically*, by which a circuit design is transfered directly onto silicon and then evaluated. Intrinsic evaluation has become feasible due to advances in recent years in PLD technologies, and has been applied to a wide range of DSP applications such as adaptive, online data compression [23]. Both intrinsic and extrinsic evaluation approaches have their advantages, each suited to different applications, a taxonomy of which is presented in Chapter 2. As a result a wide range of programmable platforms have been developed which are specifically tailored for automated circuit design using EHW, and are also discussed in Chapter 2.

1.2 Finite Impulse Response Filters

Finite impulse response (FIR) filters constitute a key function of most DSP applications and are therefore typically embedded alongside other signal processing tasks which collectively comprise the overall system. The performance and portability of hardwired FIR filters are therefore important to the fast development of application specific DSP devices, such that an existing FIR architecture can be ported into a new application with minimum re-design and test overhead.

General purpose programmable DSP chips, such as the TMS320 series from Texas Instruments, are generally not suitable for high speed FIR filtering, due to their single multiplier architecture. However, their programmability makes them suited for filter applications where flexibility of the filter for a range of applications is the primary design constraint. Programmable logic devices such as those from Xilinx [24] are suitable for implementing dedicated, high performance FIR filters. Filter functionality can also be modified, although this requires implementing a new circuit configuration on the PLD which must first be developed by the design engineer. Both programmable DSPs and PLD devices contain considerable silicon redundancy as they

provide additional functionality not required by an FIR filter architecture. An example of this is the logarithmic and exponential functions present in most DSP chips. Optimal filter performance is often only achieved through custom ASIC design. However, this is at the expense of increased design time and loss of flexibility in modifying the architecture once the device is fabricated. Regardless of the platform on which the filter is implemented, the multiplier stage is both the most complex and costly component to implement.

Evolvable hardware has already been applied to the design of FIR filters. Applications have primarily focused on the optimisation of filter coefficients, which target the multiplication stage of the filter used in coefficient generation. The multiplier is widely recognised to be the primary performance constraint when implementing the FIR filter structure in hardware. Multipliers are costly in terms of area, power, and signal delay. Several design techniques aim to reduce FIR filter complexity, and improve filter performance by targeting the multiplier. EHW techniques include the utilisation of coefficient encoding schemes designed to minimise the physical area of the coefficient multiplication stage [25]. Coefficient optimisation and hardware minimisation using EHW have also been investigated using 'multiplierless' filter architectures. This approach replaces the multiplier with a series of addition and shift operations reducing filter area at the cost of flexibility [2, 26]. Both software simulation and intrinsic hardware evaluation have been used to determine filter performance, and each optimisation approach has its benefits. Each of the EHW platforms referenced have been designed to implement the optimisation technique employed by the cited author. However, it is unclear which evaluation technique might be best suited for the hardware optimisation of a wide range of FIR filter applications, and also which programmable platform and filter optimisation approach might be most effective.

1.3 Contribution

The work presented in this thesis provides an investigation into a number of novel programmable architectures specifically developed to autonomously implement digital FIR filters using evolvable hardware. Each programmable architecture is distinguished through a number of unique characteristics, which include topologies for programmable interconnect, various input and output configurations, and the level of programmable functionality available to the architecture.

This has led to the development of a novel programmable architecture capable of implementing

digital circuits using various levels of component functionality, provided by two distinct digital logic libraries. The architecture is therefore capable of examining the relationship between the level of component functionality available, and the success of EHW to generate a circuit solution which operates under realistic physical circuit constraints [27, 28].

Particular emphasis has been placed on the coefficient multiplication unit of the FIR filter, resulting in the development of two novel programmable architectures which replace explicit multiplication with a distributed series of bit-shifts, additions and subtractions, which must be successfully configured using EHW through an integrated evolutionary algorithm [29–32].

1.4 Thesis Outline

This thesis therefore focuses on the automated design of coefficient multiplication hardware in digital FIR filters, designed using evolvable hardware. The objective of the research presented in this thesis is identified in the following statement:

A programmable architecture tailored for evolvable hardware can be developed which is highly suited to the autonomous implementation of digital FIR filters.

This thesis is organised as follows:

Chapter 2 introduces the concept of evolutionary algorithms and how they are applied to evolvable hardware. The concepts of gate-level and functional-level evolution for automated circuit design are also introduced. Extrinsic and intrinsic evaluation techniques are examined, and example applications are identified for a range of DSP applications, including FIR filter design.

Chapter 3 presents the first of three custom EHW platforms. This first platform is termed the 'Virtual Chip'. The architecture of the Virtual Chip is discussed and a number of DSP circuits are developed on the platform using both gate and functional-level evolution. Performance comparisons between both approaches are made in terms of each circuits operational speed and physical area. Further performance comparisons are made with functionally equivalent DSP circuits generated using standard design methodologies, and similar DSP circuits developed via EHW from other published works.

Chapter 4 presents an overview of FIR filter theory. A detailed examination of reduced com-

plexity FIR filter design techniques is also given, and includes an overview of multiplierless filter architectures. The primitive operator filter (POF) design methodology is then identified and its advantages, limitations and applicability for design automation using EHW evaluated. Finally an overview of general purpose programmable logic devices is presented, particularly focusing on how these devices perform multiplication and implement digital FIR filters.

Chapter 5 details the development of a Programmable Arithmetic Logic Unit (PALU) inspired by the POF approach. The development of a custom built genetic algorithm is also presented, which together with the PALU forms the backbone of the final two EHW platforms to be investigated.

Chapter 6 presents the final two programmable platforms, one inspired by the general purpose FPGA architecture, the other by a standard PLA design. Both are tailored for FIR filter coefficient multiplication using primitive operator components, and designed to be configured using EHW. Various interconnect and coefficient output topologies are examined for each of the two platforms so as to determine the most effective programmable architecture for coefficient generation. An examination of the role of mutation and crossover in automated FIR filter design is also presented. A complex and challenging FIR specification is introduced as a benchmark and used to determine the performance of both the PLA and FPGA-based platform based on a number of criteria.

Chapter 7 considers the design issues and compromises associated with translating the PLA architecture into a synthesisable netlist. Physical issues such as timing, interconnect density and drive strength are investigated. The netlisted PLA model is then compared with the original PLA architecture and evaluated using the same performance criteria identified in chapter 6. Finally, Chapter 7 investigates the ability of the PLA based EHW platform to self repair in the context of safety critical applications, or hostile environmental conditions.

Chapter 8 Conclusions obtained from each of the previous chapters are analysed, and the thesis statement critically evaluated. Further improvements are suggested concerning the design methodology behind all three EHW platforms, and the scope of the comparative analysis between platforms critically appraised.

Chapter 2
Evolutionary Algorithms for Automated Digital Circuit Design

2.1 Introduction

Ongoing advances in silicon technology continue to yielded faster and more powerful microprocessors which have enabled design engineers to examine new methods of generating circuit structures. One such method has become known as evolvable hardware (EHW), which considers the automated design of electronic systems using both software simulation and programmable hardware technologies. This chapter presents a synopsis of evolutionary algorithms (EAs), and introduces the reader to the basic concept of each class of EA, along with example applications. The role of the *genetic algorithm* as the dominant approach taken in EHW is discussed in detail, and its adaptation for automated circuit design presented. The concepts of gate-level and functional-level evolution, reflecting the granularity of logic element used in EHW to generate more complex circuit structures, is discussed. In addition, a detailed overview of circuit applications generated using both gate-level and functional level EHW is presented, and is segmented into EHW platforms which use either extrinsic or intrinsic approaches to circuit evaluation. Finally, the effects of component granularity, and the choice of encoding used to describe a circuit architecture are discussed, and their influence on the success in autonomously generating circuits using EHW presented.

2.2 An Overview of Evolutionary Algorithms

Evolutionary algorithms (EAs) are a class of non-heuristic optimisation techniques which utilise the concept of Darwinian evolution to progressively generate a given solution for a specified application. A *chromosome* describes a potential solution which intern expresses the solution's *phenotype*, or functionality. The solution for a given application lies within a *search space* which must be successfully navigated by the EA. A *population* of competing solutions concurrently investigate the search space, and are subject to selection and modification by the EA in

order to both remove poorer solutions from the search, and generate better solutions than are currently in the population. Solutions are modified through *genetic operators* termed *mutation* and *crossover*. Mutation acts on an individual by altering it in some predetermined manner, such that the resulting solution is potentially improved. Mutation is therefore analogous to a copying infidelity. Crossover usually works by combining the characteristics inherent in multiple solutions, with the aim of generating new *offspring solutions* better than the original solutions which created them. This process of evaluation, selection and modification through genetic operators is iterative, where each iteration is termed a *generation*, and is continued until an acceptable solution is found, or a specified number of generations have been performed. The terminologies used within the field of evolutionary algorithms are designed to reflect the conceptual similarity that exist between, natural evolution, EAs, and genetics.

2.2.1 Evolutionary Programming

Evolutionary programming (EP) was first developed my Lawrence J. Fogel in the early 1960's as an alternate method of artificial intelligence. Fogel's initial experiments were on the simulated evolution of finite-state machines, originally proposed by Moore [33], and Mealy [34]. Fogel's research on this subject can be found in [35].

Genetic algorithms (GAs) traditionally operate on a solution phenotype by using genetic operators to build together smaller sub-solutions or *genotypes* within the chromosome (as with biological genetic models). However, EP differs from the GA approach as it operates entirely on the phenotype, manipulating the behavioural characteristics of a solution in order to produce new offspring. As a result the quality, or *fitness*, of a given solution cannot be disseminated into smaller sub-solutions as with GAs, but is instead evaluated as a single entity, which must provide the solution required. Crossover is therefore not used in EP, which instead relies solely on mutation to affect the generation of new solutions. Competing solutions in a population are selected probabilistically such that poorer solutions within the population have a small but non-zero probability of selection. Selected individuals are then modified through one or more mutation operators, each with a specified probability distribution. This process is repeated until a new population of solutions is generated consisting of offspring solutions, and potentially a number of the original "parent" solutions. The ratio of offspring to parent solutions is determined using one of a number of population selection rules detailed in Section 2.2.2 below.

Evolutionary programming primarily uses real numbered (floating point, or integer) represent-

ations for encoding a solution in a chromosome. The choice of representation depends greatly on the application domain, and both are factors which determine the type of mutation operator required for the algorithm to generate solutions. EP has been applied to a wide range of tasks including automated control systems [36, 37], game theory [38, 39], the optimisation of neural networks weights and their structure [3–5], and route planning [40] to name but a few.

2.2.2 Evolutionary Strategies

The field of evolutionary strategies (ES) was originally developed in 1964 by Peter Bienert, Ingo Rechenberg, and Hans-Paul Schwefel, as a means of minimising the total drag of three-dimensional bodies in turbulent air flow. A number of revisions to the original algorithm have occurred since 1964, and include an increase in population size from one parent solution generating one offspring solution, to μ parents generating λ offspring. The relationship between μ and λ can be expressed by two population selection rules. The first, (μ, λ), denotes an ES that produces λ offspring from μ parents, thereby completely replacing the parent population with λ offspring. In order to maintain the population size λ must be $\geq \mu$. If $\lambda = \mu$ then the ES simply resembles a random walk search. Therefore, the relationship between offspring and parent solution in an ES is defined as $1 \leq \mu < \lambda < \infty$. This approach is often termed generational. The second rule is termed $(\mu + \lambda)$ denotes an ES that produces λ offspring from μ parents, and selects the new population from the union of both parent and offspring solutions.

ES utilise both crossover and mutation in order to generate new offspring solutions. Like evolutionary programming, ES predominantly use a real numbered encoding to describe a solution. However, unlike most non-adaptive approaches using EP, evolutionary strategies vary the individual mutation distribution of each solution according to a step size σ, based on a number of individual parameters determined by a predefined "success" rule. For example, Rechenberg's well known method of determining σ is named the 1/5 success rule [41]. This approach increases σ by a predefined value if the relative frequency of successful mutations over a specified number of generations is greater than 1.5. If this is not the case then σ is decreased by the same predefined amount. Detailed explanations of ES can be found in [42].

2.2.3 Genetic Programming

Genetic programming (GP) is conceptually very similar to GAs, and was first described by John R. Koza in 1989 [43]. However, there are a number of fundamental differences in the way that GP both encodes and manipulates solutions compared to GAs. Genetic programming uses tree-structures to evolve computer programs which describe a solution's functionality. This approach stems from the initial use of the LISP programming language for genetic algorithms presented by Cramer in 1985 [44]. Since then GP algorithms have been developed using a wide range of computing languages such as C and C++. Genetic programming commonly encodes each solution using a combination of two node classes, termed terminals and functions. Terminals are either numeric constants or other inputs external to the evolving program. Functions perform operations which take either terminal outputs, or outputs from other function nodes in order to produce new outputs which form part of the evolving solution. An example of a simple genetic program encoding the logic functions of a 2-bit multiplexor is presented in Figure 2.1. Node functions include AND, OR and NOT boolean expressions; terminal nodes represent the two inputs of the multiplexor, IN0 and IN1, and the control signal, cntrl.

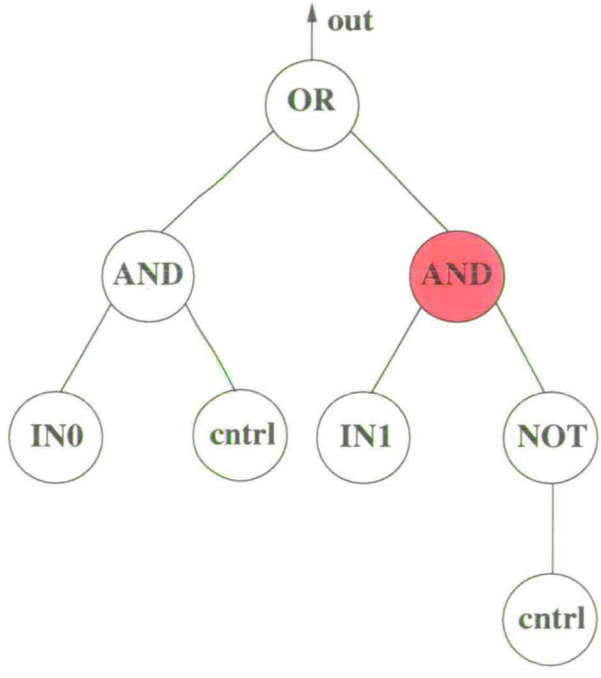


Figure 2.1: *Example of 2-bit multiplexor represented using genetic programming tree structure.*

Crossover and mutation operators are applied at selected nodes within the program tree after

the selection of fit individuals. Again the type of mutation operator employed is application specific, but usually results in the changing of a nodes functionality within specified parameters. For example in Figure 2.1 above, the NOT gate node might be replaced by an AND function as a result of mutation. Crossover acts on two selected parent programs by selecting nodes within each parent which form 'branches' of the programs encoding. In Figure 2.1 if crossover were to select the right AND node, shown in red, then the associated program branch would contain the NOT function and terminals IN1 and cntrl. These branches of program are then used to create a new offspring program which combines the selected parts of each parent. From this procedure new solutions are generated. Another important difference of genetic programming over traditional GAs is that the length of chromosome use to encode each program is not fixed during evolution. This enables a GP program to grow autonomously in order to accommodate variations in problem size and complexity. However, without careful control parameters, the length of GP tree-structures can quickly become unwieldy.

Genetic programming has been applied to a wide range of applications including the design of analogue electronic circuits [45, 46], classification of medical data [47, 48], and the automated optimisation of chemical structures [49]. A more detailed account of genetic programming can be found in [50, 51].

2.2.4 Genetic Algorithms

The concept of the genetic algorithm (GA) was first proposed by John Holland in 1975 [52], and was the first evolutionary algorithm to encode possible solutions using binary bit-strings. The prominent role of crossover to generate new variations was also introduced as an integral mechanism of the GA. Further work by Golderberg [53] supported Holland's notion of using crossover to create useful building blocks, termed schemata, which combing to form a description of a given solution. Each chromosome generated by a GA describes specific aspects of the solution it encodes. Mutation was therefore employed as a background operator, ensuring new material is randomly inserted in to the population to avoid the search stagnating around a sub-optimal solution.

The manner in which a GA both encodes and manipulates possible solutions within the search space has made it extremely suitable as the primary engine behind evolvable hardware. For example each building block (schemata) can be represented as one or more circuit components with associated connectivity. This thesis therefore focuses on the use of genetic algorithms

in evolvable hardware for the automated design of a performance driven FIR filter coefficient multiplication circuit. Genetic algorithms and their associated terminologies when applied to EHW are described in detail in the following section.

2.3 Genetic Algorithms in Evolvable Hardware

The automated design of digital circuits using EHW is not trivial. Each possible circuit solution for a given task lies within a search space. The search space is defined by the number of different component building-blocks available, the number of logic cells used to generate the circuit, and the application for which the circuit is being evolved. Genetic algorithms, when applied to EHW, perform a non-heuristic search through a space of possible circuit configurations, in order to find a solution which corresponds to a desired specification. Each circuit configuration is encoded within a *chromosome* which expresses circuit functionality. The encoded chromosome is termed a *phenotype* as it comprises numerous smaller logic cells or *genotypes*. Many possible circuit solutions are investigated concurrently, comprising a *population* of competing chromosomes. Each chromosome is evaluated and assigned a *fitness* which is representative of the quality of the circuit solution it describes. Selection mechanisms are then used to identify the most successful chromosomes within the current population. The logic elements which make up the circuits of the selected chromosomes are then modified via two *genetic operators*: mutation and crossover, to produce a new set of *offspring solutions* potentially better than their *parents*. This iterative process is repeated until an acceptable solution is found, or a specified number of iterations has been completed, where each iteration is termed a *generation*. Figure 2.2 displays the algorithmic flow of a standard genetic algorithm for evolvable hardware.

Genetic algorithms traditionally utilise a binary representation of fixed length N, which encodes a solution in a specific chromosome. Each bit in the string is termed a *loci*. Groups of loci of length L form *schemata*, which can be represented as 0, 1, or #, where # is a 'wildcard' matching either 0 or 1. Schemata therefore define useful vectors in the search space (sub-blocks of solution), which partly form the final solution. The larger the length of schema L, the greater its contribution to the final solution. The order of a schema is defined by the number of non-# loci in the string. For example, the string #10#1#0# is an order four schema, with defining loci at locations 2, 3, 5 and 7. Strings such as "01001000" and "11011101" contain the defining loci which describe the above schema. A schema's defining length is the distance between the first

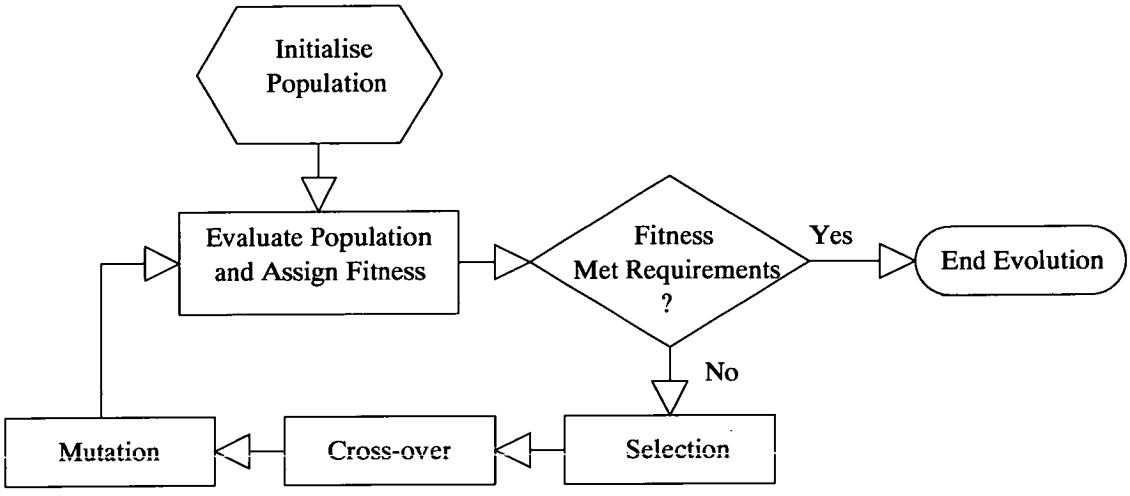


Figure 2.2: *Algorithmic flow of genetic algorithm*

and last positions of non-# loci. In the case of the example given above, the first non-# loci is at 2, and the last at 7. The defining length of the above schema is therefore 5, despite an actual schema length L of 8.

The choice of binary encoding depends greatly on the optimisation problem. If a binary representation is used to encode numerical parameters requiring optimisation, then empirical studies have found that Gray encoding is generally superior to a standard power-of-two binary coding [54]. For example, if part of a chromosome encodes a value of 7, or "0111", in a 4-bit standard binary string, then for the same string to express the value 8 all bits must change to encode the string "1000". This requires significant manipulation of the bit-string on behalf of both the crossover and mutation operators. However, as Grey coding only requires a one bit change per decimal increment, 7 and 8 can be represented as "0100" and "1100" respectively. Such a transition can easily be achieved through mutation.

Not all GA encoding schemes use bit-string representations, *real-valued* representations have also been used to encode solutions within a chromosome, often when the problem requires the optimisation of control parameters which are time or frequency dependent [54]. Generally, modified genetic operators are required from those used in binary representations in order to provide an effective means of navigating the search space. A detailed review of genetic algorithms and chromosome representations is beyond the scope of this thesis, however the reader is referred to the following for more information [42, 53, 54].

2.3.1 Initialisation

In order to begin the automated design procedure, a population of circuit chromosomes must first be *initialised*; this determines each chromosomes original circuit configuration. A number of initialisation techniques have been employed in EHW, many include heuristic knowledge of the circuit to be generated [55]. This approach is termed *population seeding*, and can be effective in reducing the number of generations needed to determine an acceptable circuit solution. One potential draw back of population seeding is that it can cause the search to become fixed around a sub-optimal, or unsatisfactory solution. This is often caused when a population of chromosomes prematurely converges around a single sub-optimal solution (circuit configuration). Seeding the population can therefore tip the delicate balance that exists between exploitation of promising solutions, and exploration of the solution search space. Aggressive selection methods, high crossover rates, and population seeding are all methods of biasing the search towards the current fittest solution in the population, thereby exploiting desirable characteristics present in the chromosomes circuit encoding, which can then proliferate throughout the population via crossover and mutation. This can therefore lead to the fixation of a population around a sub-optimal solution. A population is said to have reached *stasis* when no improvement in the fitness of a solution is experienced over an extensive number of generations.

Another requirement of population seeding is that specific knowledge of both the problem domain, and the platform on which the circuit is to be autonomously configured, is required. There are a number of cases in EHW when this information is not available, or might limit the success of the search; for example, evolving a parallel multiplier through EHW might be achieved by seeding the initial chromosomes in the population with design rules from well known circuit configurations such as the Booth multiplier. This might increase the speed in which a solution is found, however, it could also severely limit the novelty of any *new* multiplier architectures that might potentially be developed. In order to maintain the general applicability of EHW to a wide range of automated circuit design applications the most common initialisation procedure simply generates a random circuit configuration for each chromosome in the population. This approach is termed *random initialisation*, and is an accepted way of eliminating many of the undesirable effects associated with population seeding.

2.3.2 Selection

Selection is the operator used within genetic algorithms for guiding the search towards a desirable solution. The primary objective of the selection operator is to highlight better solutions in a population. Selection therefore removes poorer solutions from the population, leaving the remaining solutions available for modification through crossover and mutation operators. GAs can be classified as either *generational* or *steady-state*. A steady-state GA usually produces only one or two new solutions or *offspring* in each generation. Solutions from the current, or *parent*, population are usually deleted, based on some distribution, to make room for the new offspring solutions. Generational GAs instead replace the entire parent population with an entirely new population of offspring solutions each generation. This approach is denoted (μ, λ), where μ represents the parent population, and λ the new offspring population. Evolutionary algorithms utilise one of four selection operators.

Proportionate selection identifies individual solutions based on the proportional fitness of that individual with respect to the fitness of all other solutions in the current population. Therefore a solution having twice the fitness of another solution is twice as likely to be selected. Multiple copies of a fit solution can therefore be selected and used to form the next population. The most simple form of proportionate selection is termed roulette-wheel selection, where each solution in the population is assigned an area on the wheel which is proportional to its fitness. The roulette-wheel is then spun a number of times equal to the population size. An individual is then selected based upon where the conceptual 'marker' points. However, the basic proportionate selection operator has two important disadvantages. If a population contains a solution which is markedly superior to any other solution in the current population then a large area of the roulette-wheel will be dominated by this individual. As a result the dominant solution will most often be selected each time the wheel is 'spun'. This could lead to a reduction in solution diversity, and potentially result in the convergence of the population around a sub-optimal solution. The second disadvantage occurs when most of the solutions in the population have very similar fitness. In this case, each solution posses a roughly equal proportion of the roulette-wheel. This can have the effect of random selection, and the search then looses direction. Both these difficulties can be avoided using a scaling scheme such as that outlined by Goldberg in [53].

Tournament selection does not suffer from the disadvantages highlighted using proportionate selection. This is because selection is based on the absolute fitness of each solution in the

population, as a tournament of t individuals are randomly chosen from the current population, and the fittest solution selected. The approach therefore eliminates the negative selection bias towards highly fit solutions, whilst ensuring that the fittest individuals are continually identified, even when all solutions in the population have similar fitness. Tournament selection is also amenable to fast implementation (for example in hardware) as only a few solutions are required to be compared at a time without needing to calculate the average fitness of the population as is the case with the proportionate selection approach. The most common size of t is two, and is termed binary, or two-way tournament selection.

Ranking selection operates in a similar manner to proportionate selection except that solutions are ranked in a ascending or descending order of fitness, depending on whether the problem requires maximisation or minimisation. Each solution is then assigned a ranked fitness based on its rank within the population. Individual solutions are then selected based on a selection probability calculated using the ranked fitness score.

Boltzmann selection assigns a modified fitness to each solution based on a Boltzmann probability distribution: $D_i = 1/(1+exp(F_i/T))$, where T is a parameter analogous to the temperature term in the Boltzmann distribution. T is reduced in successive generations. Because a large value of T is initially used, almost any solution is equally likely to be selected, but, as the generations progress, T becomes small and only good solutions are selected.

Elitism can be applied to any of the selection operators detailed in this subsection. Elitism preserves solutions from the parent population and places them, untouched, into the new offspring population. This approach is used to maintain a specified number of superior solutions which might be lost during a selection. Elitism is therefore commonly used in generational (μ, λ) genetic algorithms which normally remove older genetic material from the new population. Improper use of Elitism can lead to fast convergence around sub-optimal solutions as elite individuals may be present in the population for many generations if no better solutions are found, biasing the search towards the elite individual. Retaining more than one elite solution in the population at any one time can help maintain diversity, as does limiting the number of generations an elite solution may be present in the population (its lifespan). Solution diversity using Elitism can further be maintained by using larger population sizes and higher mutation rates.

2.3.3 Crossover and Mutation

Crossover and mutation are the operators through which new circuit solutions are generated. Crossover swaps material between two circuit configurations at randomly chosen points along the chromosome, producing new *offspring*. Chromosomes of fixed bit-length must therefore cross at the same loci on both parents. The greater the number of crossover points, the higher the degree of intermixing between the two *parent* solutions. The resulting offspring encode two new circuit solutions, potentially better than the two parent solutions which generated them. The probability that two selected solutions will crossover is termed the *crossover rate.* One-point, two-point and uniform crossover are most commonly used. Uniform crossover randomly swaps individual bits between the two parents with the same probability. Uniform crossover is the most disruptive of the crossover operators, as larger schemata which make up each parent are generally not expressed in the offspring solutions. Single point crossover is used most dominantly in EHW applications which utilise genetic algorithms, and is illustrated in Figure 2.3.

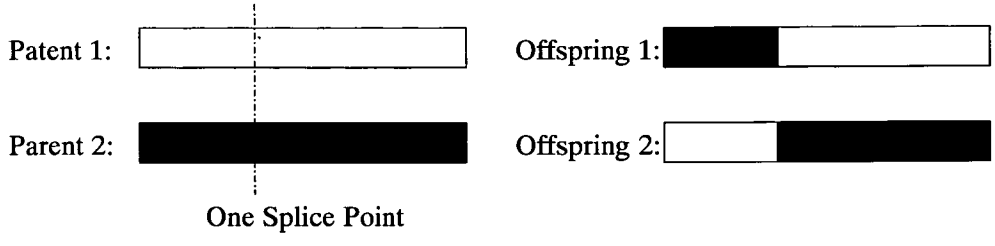


Figure 2.3: *Single-point crossover of two parent chromosomes, generating two offspring*

Mutation is used to maintain diversity within the population. It operates directly after crossover and is analogous to a copying infidelity as material is transfered from parent to offspring. Mutation is invoked with relatively low probability so as not to prove deleterious to the search. For this reason mutation is considered by Goldberg and others to be a background operator with crossover cultivating useful schemata from both parent solutions, and mutation ensuring diversity in the population through operations such a bit-flipping [53]. Bit flipping is the standard mutation operator used with bit encoded genetic algorithms, and is depicted in Figure 2.4. Each bit in the string has a probability that it will flip to its current inverse (i.e 0 to 1); mutation rates are usually set such that on average one mutation occurs per chromosome (bit string). Therefore a chromosome of bit length 100 would have a mutation rate of 0.01. Mühlenbein [42]

expresses this relationship more formally as

$$P_m = \frac{1}{N} \tag{2.1}$$

Where P_m is the probability of a bit mutation within a chromosome of bit length N.

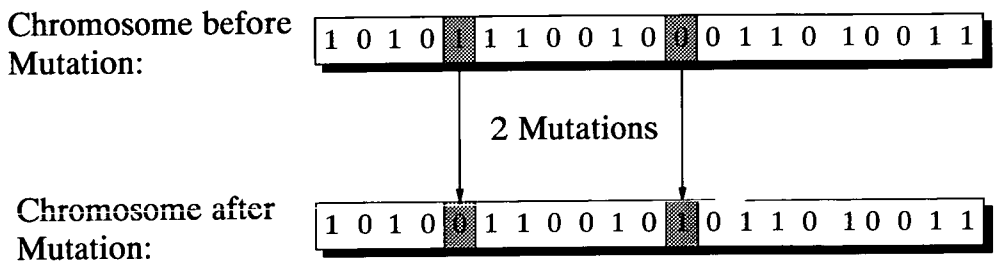


Figure 2.4: *Bit-flipping mutation example in bit-level chromosome encoding*

2.3.4 Fitness Function

Development of the fitness function in EHW is directly related to the circuit application. The success of the GA in finding an acceptable circuit solution is therefore most influenced by how accurately the fitness function describes a given circuit specification. When using EHW for automated circuit design, fitness is often represented as a percentage of circuit functionality. Correctness if therefore frequently calculated by summing the total number of correct bits produced by the circuit under evaluation and comparing this to the desired output response. For example, consider the look-up table for a 2-bit parallel multiplier as presented in Table 2.1. Both *Ain* and *Bin* are 2-bit input vectors, therefore the total number of 4-bit test vectors required to ensure correct functionality are: $V_{test} = 2^I$, where I is the total number of bits applied to the circuit input. In this case $2^4 = 16$ 4-bit test vectors are needed to fully test the multiplier. A total of $2^4 * 16 = 256$ 4-bit output vectors are therefore possible, from this the correct circuit configuration which produces the desired sixteen 4-bit output vectors must be identified. Fitness might then be calculated by comparing the number of correct output bit vectors, produced by a given circuit solution, i, as shown in equation 2.2.

$$F_i = O_i/V \tag{2.2}$$

Ain	Bin	Multi out
00	00	0000
00	01	0000
00	10	0000
00	11	0000
01	00	0000
01	01	0001
01	10	0010
01	11	0011
10	00	0000
10	01	0010
10	10	0100
10	11	0110
11	00	0000
11	01	0011
11	10	0110
11	11	1001

Table 2.1: *Boolean logic look-up table of 2-bit parallel multiplier*

Where O_i is the number of correctly matching output bits for the current solution, and V is the total number of bits which must be matched. The simple 2-bit multiplier example above provides an indication as to both the size and complexity of the search space which must be successfully navigated if EHW is to prove an effective tool for automatically designing industrially useful DSP circuits.

2.4 Applying Evolvable Hardware to Automated Circuit Design

Circuits generated via evolvable hardware are evaluated by one of two methods: *extrinsic* evaluation (software simulation), and direct *intrinsic* evaluation by which a circuit is transfered directly into silicon and then evaluated. Intrinsic evaluation has become feasible due to recent advances in the past decade in programmable hardware technologies such as PLDs (*Programmable Logic Devices*) and FPGAs (*Field Programmable Gate Arrays*), both of which are detailed in chapter 4.

With the advent of faster and larger FPGAs, resulting from advances in silicon technology, and the move towards deep sub-micron technologies, designers are under increasing pressure to provide high performance DSP circuits which take advantage of these new platforms. *The*

result are circuits which must operate under critical constraints imposed by high density, and the domination of interconnect capacitance [56].

The successful generation of digital circuits through extrinsic and intrinsic evaluation has demonstrated the great potential of EHW for automated circuit design. It has also raised a number of questions as to how current EHW techniques can be further improved.

2.4.1 Gate Level and Functional Level Circuit Evolution

Evolvable hardware generates circuits by manipulating a number of 'building blocks' which the EA has at its disposal. When applied to digital circuit design these building blocks generally take the form of *primitive gates*; basic logic elements such as ANDs, NOTs, ORs and XORs. EHW architectures which use primitive logic elements are said to perform *gate-level* evolution. One problem with gate-level evolution is that encoding lengths can become unwieldy when larger circuits are to be evolved. However, Thompson [13] argues that fine-grained building blocks such as these enable EHW to develop novel architectures beyond the scope of conventional design techniques. Despite this, evolving with primitive gates is thought to impede the success of EHW frameworks in evolving all but relatively simple DSP circuits [57]. Miller et. al. have demonstrated the evolution of 2-bit, 3-bit and 4-bit parallel multiplier circuits using gate-level EHW [12, 58], reflecting some of the most complex DSP circuit currently generated with the gate-level EHW approach. However, multiplier complexity greater then 4-bits has not been achieved using gate-level evolution. An alternative to gate-level evolution is that of *functional-level* evolution [59, 60]. Here larger logic elements, or macros, are used comprising many primitive gates. An example of a functional-level component within a multiplier design might be a fulladder, or half adder circuit. This approach is further investigated by Ersin in [61], in which a number of macro units are utilised to evolve more complex arithmetic logic units. Function-level evolution has been shown to produce DSP circuits which are of a level of functionality sufficient to be of use to industry. These include online data compression for colour printers [62] and the adaptive equalisation of digital communication channels [63].

2.4.2 Digital Circuit Design using Extrinsic Evaluation

Evolvable hardware for automated digital design favours software based, or *extrinsic* evaluation, due to the simplicity of its implementation and the ease in which evolved circuits can be

examined once a solution is found [64]. Using this approach only the final solution is downloaded onto a reconfigurable device, or fabricated on a custom IC. The majority of frameworks which employ extrinsic evaluation use a technology independent net-list to model the digital circuit undergoing evolution. Examples include the use of genetic programming to synthesis logic functions using generic Multiplexer trees [64, 65]. Drechsler and Günther take this approach further by evolving logic functions using Multiplexer circuits which can be mapped onto Multiplexer-based FPGAs [66].

Other extrinsic platforms are more closely based on programmable logic devices which are discussed in detail in Chapter 4. Arslan et.al. presents a novel FPGA-based architecture designed to implement digital filters [67], whilst Miller and Thomson have developed a chromosome representation which more closely models the architecture of Xilinx's now obsolete XC6200 series of FPGAs [68] for evolving novel 2-bit multiplier circuits. In addition, the multiplier structures developed by Miller et.al and discussed in section 2.4.1 were implemented using a second, more constrained FPGA-based circuit encoding. This produced significant improvements over the XC6200 model, enabling the evolution of 3 and 4-bit multiplier circuits. The use of FPGA-based architectures for EHW is an important issue which will be investigated in more depth in Chapter 6.

One of the dominant limitations in using extrinsic circuit evaluation, which is common to all of the approaches cited above, is that little or no information is processed in terms of how accurately a system is modelled in terms of the circuits physical characteristics, such as timing. This is because in most cases the additional detail required has not been integrated into the software. This inhibits the development of high performance DSP circuits where timing and area constraints are of great importance, and therefore *must* be accounted for in the EHW design procedure.

2.4.3 Digital Circuit Design using Intrinsic Evaluation

Research on intrinsic circuit evolution has presented a whole new aspect of automated digital circuit design as well as fundamental problems resulting from the effects of circuit evaluation using the 'unconstrained' intrinsic approach.

Thompson's development of a two-tone discriminator using gate-level, unconstrained evolution [13] has shown that the evaluation of a circuit intrinsically in silicon can be affected by

the physics of the device upon which evaluation takes place. Circuits evolved using this technique have been shown to be temperature, silicon, and voltage dependent. It is presently unclear why intrinsic evaluation exploits a given architecture's physical characteristics, or how these negative effects can be minimised [19, 69]. Layzell has since attempted to provide an environment suitable for answering these fundamental questions by using a general-purpose evolvable motherboard consisting of an array of programmable switches, connected to up to 6 plug-in daughterboards. Each daughterboard can theoretically perform any number of functions, however, Lazell focuses on daughterboards containing arrays of operational amplifiers and transistors [70]. Circuits evolved on the motherboard included a simple NOT gate, an amplifier, and oscillator. Again each circuit exhibited dependence on the components on which they were evolved, such that if new transistor components were inserted then the system must be re-evolved to regain the desired functionality. Layzell also developed a software model of the evolvable motherboard. Results presented in [70] show that extrinsic evaluation provides solutions to each of the the circuits investigated, and could be used to configure the hardwired motherboard with minimal re-evolution.

On solution to the problems inherent in intrinsic circuit evaluation has been presented through the development of a Java-based tool for evolving gate-level circuits on Xilinx's XC4000EX/XL series of FPGA devices [71]. The architecture, called *GeneticFPGA*, uses Java to interface to the XC4000EX/XL bitstream, which then generates circuits on the fly. This EHW platform avoids the problems associated with unconstrained intrinsic evaluation by driving all inputs with flip-flops in a completely synchronous mode. In addition, evolution is constrained such that only neighbouring connectivity is possible so as to avoid possible contentious circuit configurations, such as feedback and same pin multiple source short circuits.

Tufte and Haddow implement the genetic algorithm directly on an Xilinx XC4044XL FPGA such that the GA and the evolving circuit design are implemented together on the same device. They coin this process *Complete Hardware Evolution* (CHE) [72]. Early work focused on the automated design of Multiplexer circuits, however, more recent work uses the same approach to autonomously evolve adaptive filter coefficients within a constrained filter architecture embedded on the FPGA [17]. The CHE principle has also been demonstrated earlier by Kajitani et. al in [60] and was used to implemented many of the practical applications detailed in [11].

Due to the limited commercial availability of analogue programmable devices, only a small number of architectures suitable for evolvable hardware have been developed in the academic

field. Whilst this thesis does not focus on analogue circuit design using EHW, a number of programmable devices have are now included for completeness which are capable of implementing both analogue and digital circuits using EHW. Examples include the Palmo chip developed by Hamilton et. al. in [73]. The device is constructed in mixed-signal VLSI and can process analogue signals specified as pulse streams. Palmo can therefore be used to process mixed-signal data effectively on a single programmable device. The Palmo chip has been shown to be a useful platform for EHW design techniques, where the evolution of novel filter structures has demonstrated [74]. Stoica et.al have developed a reconfigurable transistor array specifically designed for evolutionary oriented circuit design [75]. The device can implement both analogue and digital circuits and exhibits robustness to extreme temperature variations when evolved on-line.

2.4.4 Encoding Digital Circuits Using Evolvable Hardware

Gate-level evolution, using either extrinsic or intrinsic evaluation, presents a number of difficulties as circuit complexity increases. These difficulties become manifest in the encoding schemes used to represent circuit functionality. Lengthy chromosomes are attributed to the manner in which circuits are encoded using the gate-level approach [14]. As circuit complexity increases, so too does the number of logic gates required to build it. This increase in gate count relates directly to an increase in the chromosome length, as each logic gate must be encoded. Because the majority of evolutionary algorithms require populations of chromosomes to find an adequate solution, reducing chromosome length can greatly reduce the memory requirements of many EHW applications. Limiting the manner in which logic gates can connect, and increasing the building block size to accommodate function-level evolution are two methods of limiting the length of circuit chromosomes in EHW. Higuchi et.al. have demonstrated both reduction methods successfully in [59]. As an example consider the two encoding approaches used to describe the circuits illustrated in Figure 2.5. Both circuits are functionally identical, however, the gate-level encoding would require a seven cell description to represent the circuit, while the functional-level encoding would require only three. Although more cell connectivity information is required to encode the fulladder cell described using the functional-level approach, the overall reduction in chromosome length justifies this. Again, it could be argued that increased component granularity and reduced freedom of component connectivity might reduce the novelty of circuits produced using EHW. Such an argument should however also take into consideration the functional complexity of the application, and thus the size of the search

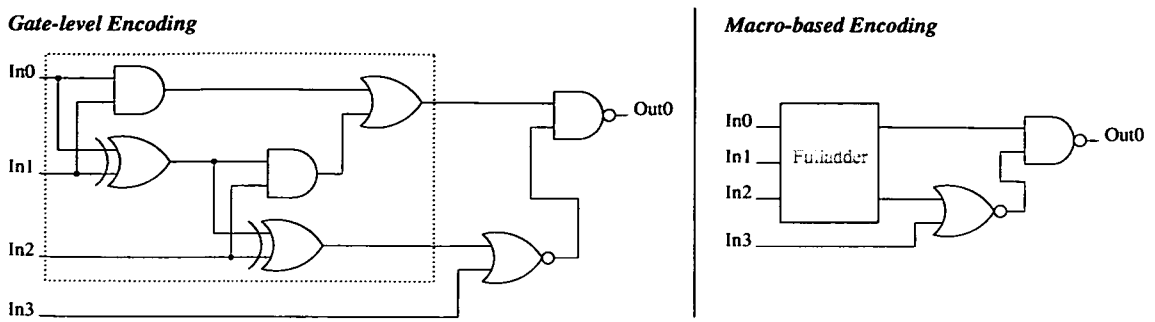


Figure 2.5: *Comparison of a standard gate-level encoding with the novel macro-based encoding to describe a Fulladder with additional logic.*

space, the memory resources available to the EHW platform, and the designers requirement for circuit novelty against speed of implementation. Chapter 3 will also show that functional-level evolution requires fewer generations to attain functionally correct DSP circuits, than equivalent circuits generated using gate-level evolution.

2.5 Summary

This chapter has introduced the concepts of evolvable hardware, and presented four derivative classes. A detailed overview of genetic algorithms and associated genetic operators tailored for EHW applications has been presented. This has highlighted the GAs suitability for automated circuit design and demonstrated how genetic operators might be used to developed circuit architectures for a given specification, often in the form of a boolean lookup table. The benefits and limitations of both gate-level and functional-level approaches to automated circuit design have been investigated through literature review. Gate-level evolution has provided a number of novel DSP circuits with smaller area than that achieved using conventional design techniques, however, a non-linear growth exists between the complexity of the circuit to be evolved and the size and complexity of the search space in which an acceptable solution might be found. This has limited the effectives of gate-level EHW to relatively simple DSP circuits. Functional-level evolution has demonstrated the automated design of a number of much more complex DSP applications, where the search space has been constrained for the specific DSP task. It has however been stated that functional-level building blocks limit the novelty and performance of circuits generated through EHW, although the author is unaware of any direct comparisons

between gate-level and functional-level approaches

The merits of both extrinsic and intrinsic circuit evaluation using EHW have also been subject to literature review. Applications requiring on-line adaptation must inevitably be implemented directly in hardware and therefore suit the intrinsic approach. This technique has the disadvantage of restricting the DSP to one technology platform, and has been shown to produce unstable circuits which are reliant on the specific physical characteristics of the device upon with the are evolved; unless necessary constraints are set in place as part of the circuit encoding. Extrinsic evaluation through software simulation provides the design engineer with a means of more accurately assessing the circuit developed using EHW. However, physical circuit characteristics such as timing are usually ignored in favour of area minimisation. The next chapter explores the development of a more accurate software simulation environment for autonomously developing DSP circuits, and provides a direct comparison of gate-level and functional level approaches to circuit design for a number of DSP applications.

Chapter 3
Generating DSP Circuits on the Virtual Chip EHW Platform

3.1 Introduction

Because of the need for high performance DSP applications, many modern designs must target DSM (deep sub-micron) technologies to achieve demands set by low area and fast data throughput. As a result timing and area issues have become a dominant factor in the design of performance DSP circuits and therefore should be accounted for by EHW platforms when designing such applications.

The EHW platform presented in this chapter was developed to incorporate performance related design criteria concerning the area, timing and correct functionality of DSP circuits. Also, the platform was developed to ascertain the merits of both gate-level and functional-level approaches to the automated design of a high performance multiplier stage for FIR filter coefficient multiplication. A number of other DSP applications are also investigated to provide a wider basis for comparison. The work set out in this chapter therefore aims to achieve the following:

- Investigate the use of a custom evolvable hardware platform, termed the Virtual Chip, to produce novel, high performance DSP circuits, developed under area and timing constraints.
- Provide a performance comparison of circuits generated using functional-level evolution with functionally equivalent circuits developed using gate-level evolution.
- Provide a basis for analysis by comparing those circuits generated by the Virtual Chip with functionally equivalent circuits developed using standard CAD-based design methodologies.

The results obtained in this chapter intend to establish the effectiveness of gate-level and functional-level design approaches to automated circuit design using a genetic algorithm and

EHW platform common to both approaches as the basis for comparison. The effectiveness of the DSP circuits produced using the Virtual Chip will also be compared with equivalent circuits generated using other published EHW platforms, and circuits developed using standard digital design methodologies.

3.2 The Virtual Chip Evolvable Hardware Platform

The Virtual Chip has been designed to provide an automated digital circuit design environment. Within this platform a novel genetic algorithm encoding is used to evolve digital circuits. Evaluation is performed extrinsically through detailed simulation of circuit functionality and timing characteristics. The Virtual Chip platform was designed to be a generic, flexible environment for generating a wide range of digital circuits. Although this is not the underlying motivation of this thesis, a flexible platform was required so that a range of DSP circuits could be *evolved* in order to determine the effect of component granularity (gate-level vs functional-level evolution) on the effectiveness of an EHW platform to develop non trivial DSP circuits. This research translates as the first stage in the development of the more complex DSP functionality needed for FIR filter design.

3.2.1 Encoding a circuit within the chromosome

Genetic algorithms for evolvable hardware are used to develop chromosomes which then encode the functional description of a given circuit. As with many applications which utilise genetic algorithms, the resulting circuit is termed a *phenotype* as it comprises numerous smaller logic cells or *genotypes*. The terminologies used are designed to reflect the conceptual similarity between genetic algorithms, natural evolution, and genetics.

The genetic algorithm presented in the Virtual Chip platform uses a permutation-based integer encoding of fixed-length. As such a specified number of logic elements are presented to the framework. From this, the desired circuit functionality must be generated. Using a fixed-length encoding is standard practice and is one of the main restrictions within which a genetic algorithm operates [53].

Specific sections of each chromosome are reserved for describing the inputs and outputs required for the desired circuit. Logic elements are referenced by position within the chromo-

some. Figure 3.1 displays the relative location of each encoded section. Circuit inputs are

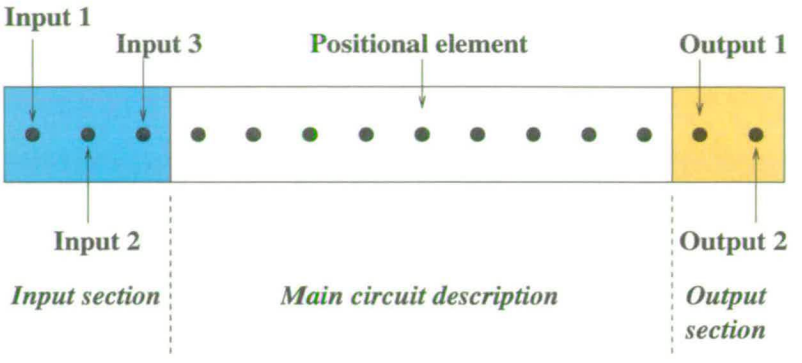


Figure 3.1: *Chromosome structure defining sections for specific circuit description*

encoded in the first section of chromosome. If a circuit has I inputs, then the first I logic elements in the chromosome will describe these inputs. This description includes the input pin number in addition to which logic element the input pin is connected. Outputs are similarly defined at the end of the chromosome, where position relates to the identification of an output pin connected to a logic element. Total chromosome length, N, is then defined as the number of logic elements summed with the number of circuit inputs. Therefore, if a circuit has two outputs, what ever logic elements are at N and $N\text{-}1$ are connected to output pin one and output pin two respectively. The encoding ensures that the number of inputs and outputs described by a chromosome remains consistent after operations such as crossover.

The GA comprising the Virtual Chip utilises a range of functional elements or *macro blocks*, along with simple gate primitives with which to generate various DSP circuit structures. Macro blocks particularly suited to more complex DSP circuits were chosen such as a halfadder and fulladder. Other macro cells include small combinational logic blocks. In addition, through-connects are provided to increase the flexibility of the circuit encoding. Table 3.1 lists all of the logic elements available to the GA and indicates if the component constitutes a primitive or functional logic element. Figure 3.2 clarifies a number of the component terminologies presented in Table 3.1, and displays examples of the type of logic element. Two component libraries are therefore available, one representing gate-level evolution, the other function-level. A total of 14 logic elements are included in the gate-level library, and 28 logic elements (primitive logic elements are also included) in the functional library.

Each cell is connected within a flexible chromosome encoding which allows placement of any

Primitive Logic Elements	Functional/macro Logic Elements
2-input NAND	3-input NAND
2-input XOR	3-input XOR
2-input XNOR	3-input NOR
2-input OR	3-input OR
2-input AND	Common XOR
2-input NOR	Common AND
BUFFER	Common NAND
NOT	Common OR
Pull-high	Half-adder
Pull-low	Fulladder
Through-connect	Combinatoral NAND
Through-connect + float	Combinatoral AND
Pull-high + float	Combinatoral OR
Pull-low + float	Combinatoral XOR

Table 3.1: *Primitive and functional logic elements available to genetic algorithm within Virtual Chip EHW platform.*

cell (functional or primitive) into any position within the string. This provides the EHW platform both the flexibility of standard gate level encodings, and the potential of building more complex systems afforded by less flexible functional-level architectures.

3.2.2 Connecting Cells Within the Chromosome

Each genotype (logic element) in a circuit is allocated a specific position within the corresponding chromosome. The type of logic cell at any given position is initially determined randomly, however cells can be allocated different positions after initialisation through manipulation by the *genetic operators*, detailed in section 2.5.3. Figure 3.3 demonstrates the chromosome encoding scheme used to describe connectivity of the fulladder cell depicted in Figure 2.5. It is important to note that a cells connectivity is not restricted to its nearest positional neighbour. Rather, cells are free to connect to any cell of higher position within the chromosome. This form of 'over-the-cell' connectivity provides a much wider range of possible circuit configurations. Feedback connections are not permitted as their effects are not desirable for most DSP applications, with the exception of functions such as the Infinite Impulse Response (IIR) filter. However, the chromosome encoding used does allow for feedback connectivity should such a feature be required. Feedback is simply achieved by allowing logic cells to connect to cells at a lower chromosome position than the current cell. Logic cells near the end of the chromosome

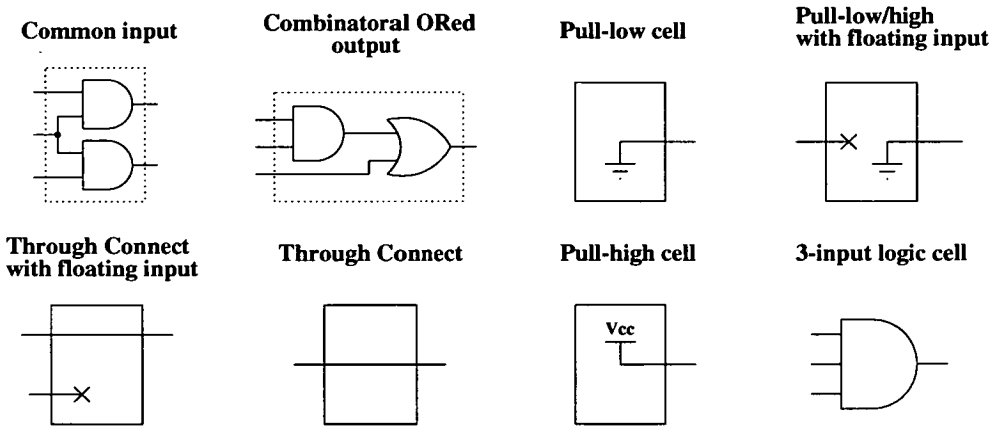


Figure 3.2: *Generic style of macro and other logic elements provided to component library for the evolution of arithmetic circuits.*

Macro-based Encoding

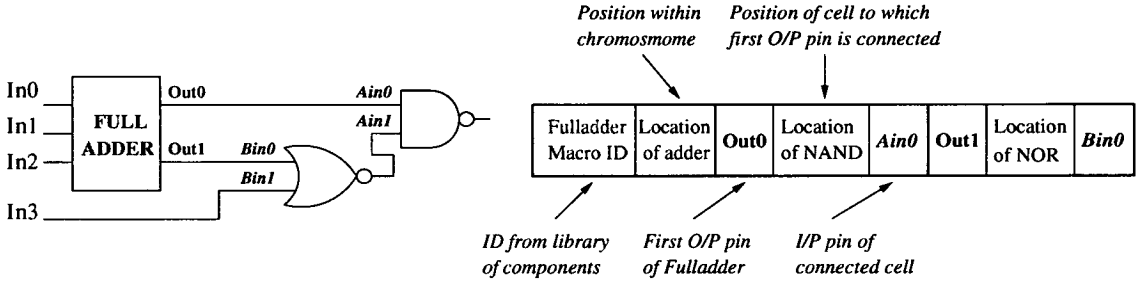


Figure 3.3: *Example of macro-based encoding describing a macro element (fulladder) and its connectivity.*

may therefore loop back there output connections to earlier cell locations.

3.2.3 The Genetic Operators

The genetic algorithm utilised by the Virtual Chip use single-point crossover. Single point crossover was used as it is the least disruptive means of combining circuit characteristics between two *parent* solutions. Because of the complex interactions between logic elements which form a circuit, the effects of crossover (and also mutation) can be highly disruptive to the search. This is often caused by the breaking of connections between logic elements due to the recombination process, and the high degree of interdependence between logic elements, termed epistasis, which is needed to achieve a desired circuit functionality. Figure 3.4 illustrates the effects of

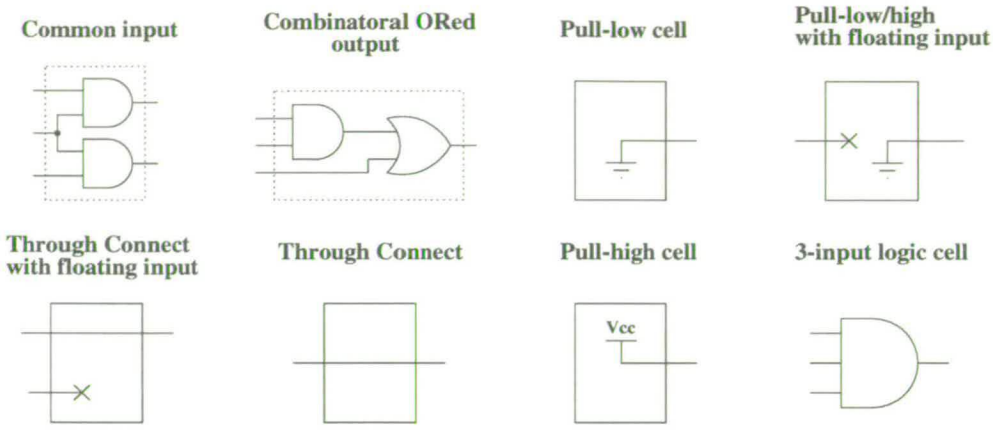


Figure 3.2: *Generic style of macro and other logic elements provided to component library for the evolution of arithmetic circuits.*

Macro-based Encoding

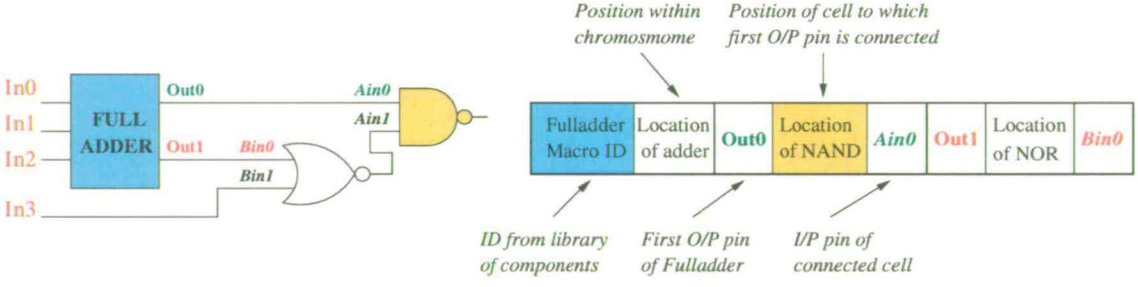


Figure 3.3: *Example of macro-based encoding describing a macro element (fulladder) and its connectivity.*

may therefore loop back there output connections to earlier cell locations.

3.2.3 The Genetic Operators

The genetic algorithm utilised by the Virtual Chip use single-point crossover. Single point crossover was used as it is the least disruptive means of combining circuit characteristics between two *parent* solutions. Because of the complex interactions between logic elements which form a circuit, the effects of crossover (and also mutation) can be highly disruptive to the search. This is often caused by the breaking of connections between logic elements due to the recombination process, and the high degree of interdependence between logic elements, termed epistasis, which is needed to achieve a desired circuit functionality. Figure 3.4 illustrates the effects of

recombination through crossover.

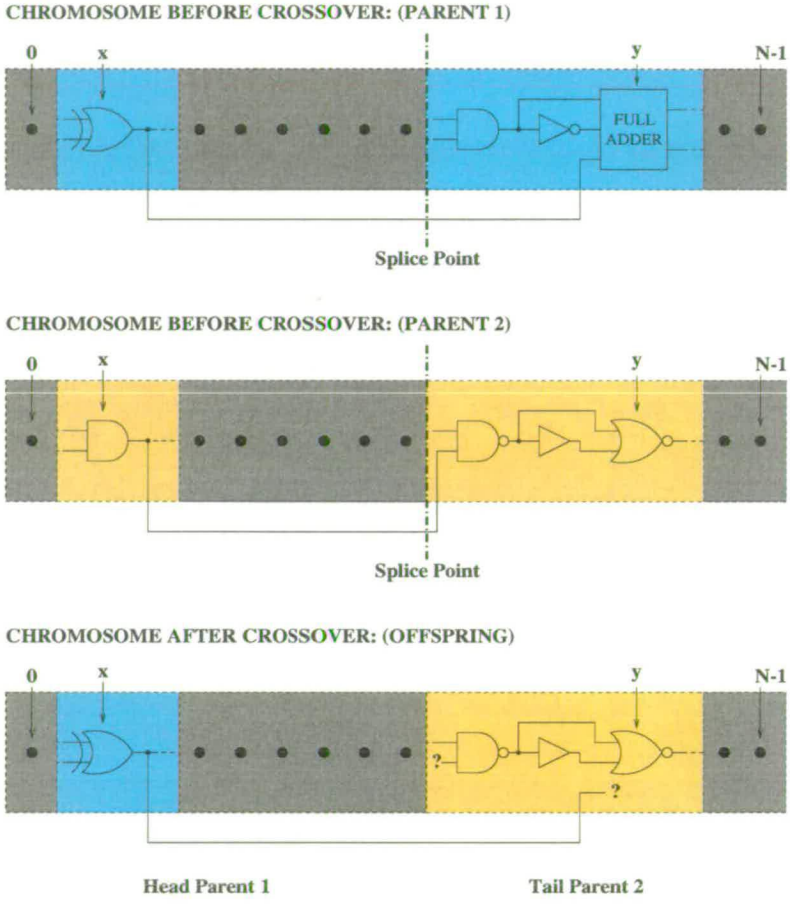


Figure 3.4: *Example of broken element connectivity resulting from crossover.*

It should be noted that unlike bit-wise crossover described in section 2.2.3, the crossover operator detailed in this chapter works in a different manner because of the integer-based encoding utilised by the Virtual Chip platform. Crossover therefore only splices each circuit encoding at the beginning of a specific section of chromosome (schema), which then describes a individual logic element and its connectivity. It is therefore not permissible to splice a chromosome mid-way through the encoding of an individual logic element. For example, if the crossover location of a parent chromosome were chosen to lie mid-way through the encoding of a Fulladder at location x, and the corresponding x location on the second parent related to the encoding of a 2-input NAND gate, then the resulting offspring circuit might encode an invalid description of a logic element, as the the physical characteristics of both the Fulladder and the NAND gate are very different.

So as to minimise the negative effects of crossover, chromosome 'repair' is used to reconnect any element connections broken during the operation. This is achieved using a nearest neighbour connection rule. It can be seen that with the example offspring chromosome depicted in Figure 3.4, the logic element at position x is no longer able to connect to the new logic element now at position y. The repair algorithm instead attempts to connect element x to its nearest neighbour at position $x + 1$. If this is unsuccessful then subsequent reconnection attempts are made from $x + 2$ to $N - 1$, where N denotes the total number of logic elements present in the chromosome. In the event that no logic elements are available for connection, the current element is assigned as floating and further attempts at reconnection made during later crossover operations. If feedback connectivity is enabled, the search for reconnection does not finish at the last chromosome position, instead the search loops back to the start of the chromosome and will continue until the logic cell immediately before the current element has been examined. This is to prevent direct component feedback which can damage the circuit. Each logic element in the resulting offspring chromosome is examined for broken connections after every crossover event.

Mutation is invoked with relatively low probability so as not to prove deleterious to the algorithm search. There are four circuit-specific mutation operators used within the genetic algorithm presented. Each is depicted in Figure 3.5.

Each operator was specifically designed to enhance the genetic algorithm by providing it with the ability to introduce both new logic elements and connections not obtainable using crossover. Of these operators only *cell replacement* needs explanation. The result of this mutation is to replace an existing logic element in the chromosome with one randomly selected from the component library. This ensures than new solutions can be obtained through diversification many generations after population initialisation.

The GAs mutation rate is a derivation of the chromosome bit-length relationship originally proposed by Mühlenbein [42], and defined in equation (2.1). Where the number of bits used to encode the chromosome, L, governs the mutation rate. Because the genetic algorithm presented here uses a permutation-based integer encoding, a direct translation of Mühlenbeins relationship is not possible. Instead the total number of logic elements encoded in the chromosome N is used to represent L. Mutation therefore operates on each logic element with the same probability rate given above. If an element becomes subject to mutation, one of the four mutation operators highlighted in Figure 3.5 is applied, each with an equal probability of selection (i.e.

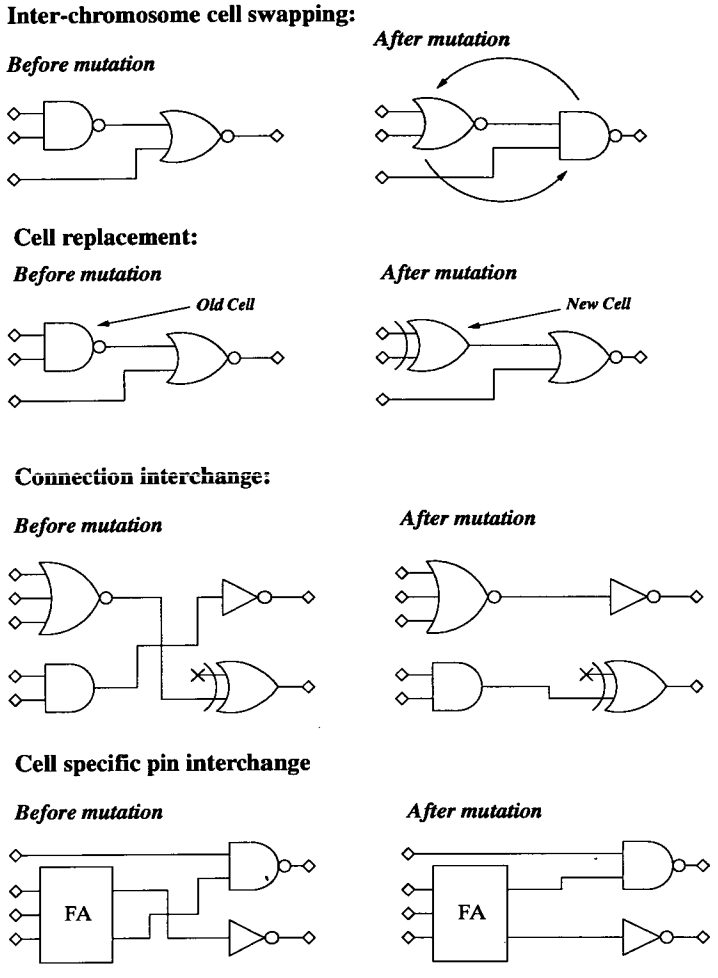


Figure 3.5: *Four mutation operators used by the genetic algorithm.*

0.25).

The GA in the Virtual Chip uses two-way tournament selection for the reasons described in section 2.3.2 of chapter 2. Whilst a number of selection methods could have been investigated, an important focus of the research presented in this chapter was to determine the effectiveness of both primitive and functional-level evolution in enabling the GA to find successful DSP circuit solutions. The selection method employed therefore simply provided a common basis for comparison.

Several constraints are imposed during initialisation of the genetic algorithm. Some are designed to eliminate contentious circuit configurations, while others are a result of the evolutionary algorithm employed. Initial global parameters are entered by the designer and are as

follows:

- Number of inputs and outputs required for the desired circuit;
- Definition of input and output vectors upon which evaluation takes place, and which describe circuit functionality;
- Number of logic elements within a chromosome used to create the circuit;
- The maximum number of possible fan-outs per cell output;
- Definition of global clock speed for timing constraints;
- Population size, defining the number of circuit solutions concurrently evolving within the search space.

A summary of the parameters applied to the genetic algorithm are as follows:

- Generational genetic algorithm
- Two-way tournament selection
- One-point crossover at 0.7 and chromosome repair applied
- Mutation using Mühlenbein derivation $P_m = 1/L$, with application specific operators
- Population size fifty

So as to optimise cell connectivity within a fixed-length circuit encoding, each output pin on a logic element is randomly allocated a fan-out ranging between one, and a user defined maximum. Fan-out describes the number of logic elements that an individual element may connect with. The connectivity of any specific logic element is not restricted to its nearest positional neighbour.

Circuit correctness is evaluated using the fitness scheme described in section 2.3.4, and calculated using the fitness expression presented in equation in 2.2. Each circuit is firstly tested through interaction with a HDL (Hardware Description Language), described in the following section. A population of 50 was chosen as it is commonly used in other EHW applications and represented the maximum number of solutions which could be evaluated without incurring a prohibitive delay.

3.2.4 Circuit Evaluation with the Virtual Chip

Hardware description languages (HDLs) are predominantly used by design engineers when developing performance circuits. VHDL (*Very High Speed Integrated Hardware Description Language*) is one of two dominant HDLs for describing digital electronic systems [76]. VHDL is a technology independent environment that describes the structure of a digital system by describing electronic subsystems (logic elements) and how they are interconnected. In addition, circuit descriptions can then be accurately simulated without the need for hardware prototyping. VHDL is therefore a powerful tool for both circuit design and evaluation, and provides an ideal environment for EHW techniques which utilise extrinsic evaluation. Few EHW platforms have been developed which utilise HDLS for circuit evaluation, however one example can be found in [77]. The technique cited however does not take into account the physical characteristics of the circuit undergoing evaluation.

After successful testing a circuit can then be synthesised to provide a technology specific netlist, ready for transfer onto silicon. A netlist describes the physical composition of a given circuit, and includes details of the type of logic component used and how it is connected. Almost all technology vendors provide models for logic elements within component libraries. The Virtual Chip therefore evolves the structure of a circuit directly within the VHDL language. This is performed within a specially designed testbench. It is this testbench which instantiates and interconnects all the logic elements within a chromosome which encodes a specific circuit solution. Evaluation is performed by instantiating and simulating all the circuits described within a population of chromosomes, as if they were being implemented within a single reconfigurable chip. Simulation of each circuit was performed using Cadence's *Leapfrog* VHDL simulation tool. Figure 3.6 illustrates a 2x2-bit multiplier evolving within the Virtual Chip environment. As can be seen in Figure 3.6, each 2x2-bit multiplier has 4 inputs and 4 outputs. All inputs and outputs are synchronised with flip-flops to account for propagation delays and ensure that all output signals have reached a steady state. It is these flip-flops which, governed by a global clock, set the timing constraints within which the evolving circuit must operate. A circuit with incorrect timing will produce output signals offset with those desired and will therefore incur low fitness.

Each 4-bit output grouping represents an individual circuit evolving within the virtual environment. Each grouping is tagged according to the circuits ID within the evolving population. Every circuit solution is therefore represented as a technology independent VHDL netlist. Netl-

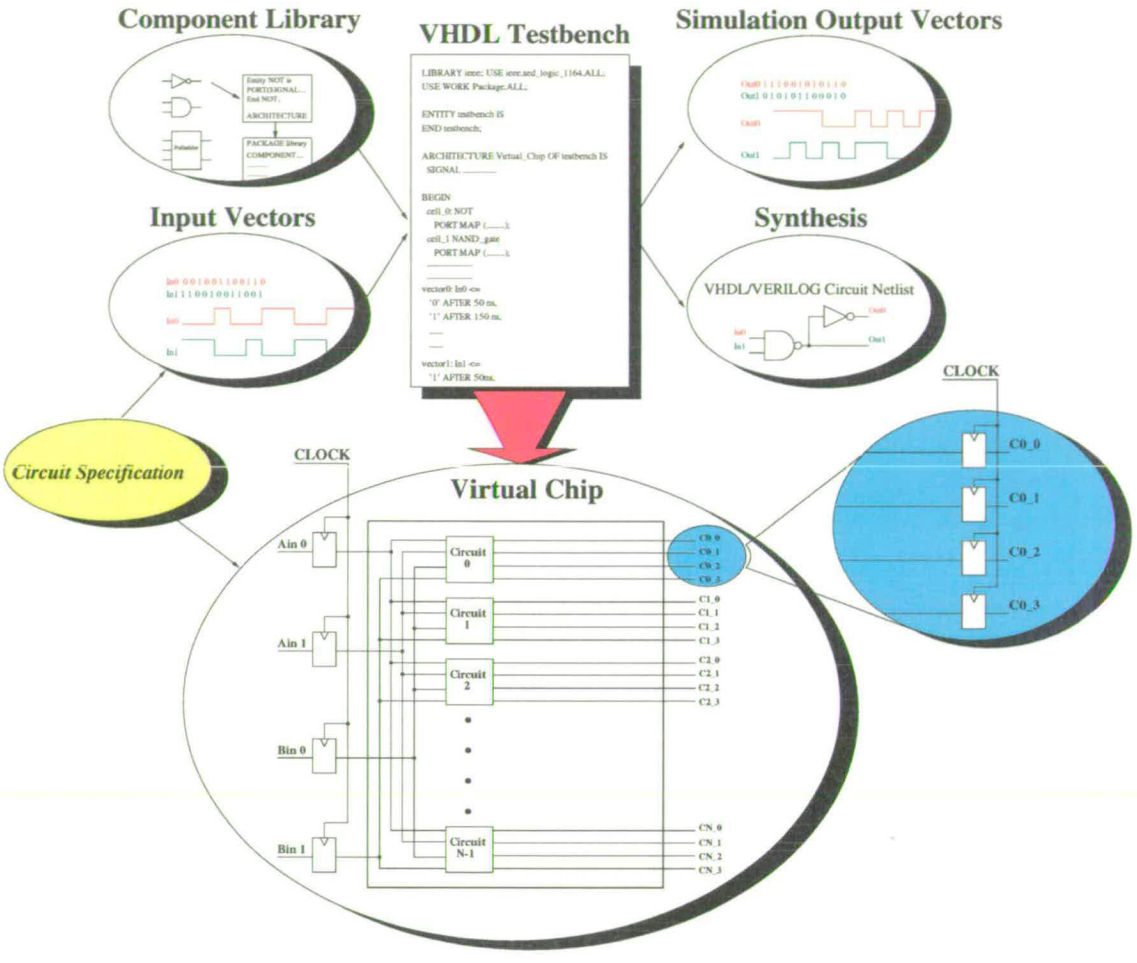


Figure 3.6: *Graphical representation of the Virtual Chip environment, evolving a 2-bit multiplier within a population size of N.*

ists are direct interpretations of the circuit chromosome and define a circuit in terms of its logic elements and inter-connectivity. Standard CAD tools can then be used to both optimise the circuit and translate the generic netlist into a technology specific netlist suitable for implementation in hardware.

Due to the *implicit* parallelisation of the Virtual Chip environment, the entire population is compiled, and simulated as one entity. This differs from the majority of EHW platforms which use extrinsic evaluation and evaluate each individual solution sequentially. As a result, within the Virtual Chip environment, an entire population of fifty individuals, evolving fifty 2x2-bit parallel multiplier circuits, can be simulated and evaluated in approximately five seconds. In contrast, if each circuit were evaluated as an *individual* entity, it would take approximately two

minutes to evaluate the same population. These figures were obtained on a standard Sparc Ultra 10 workstation with 640Mb of memory.

The Virtual chip is a fusion of C code and VHDL. The genetic algorithm itself is executed in C and generates the VHDL required to instantiate each chromosome encoded circuit. After a circuit has been successfully evolved it is then passed through a CAD tool for optimisation. Figure 3.7 displays the execution flow and coding format of the Virtual Chip EHW platform.

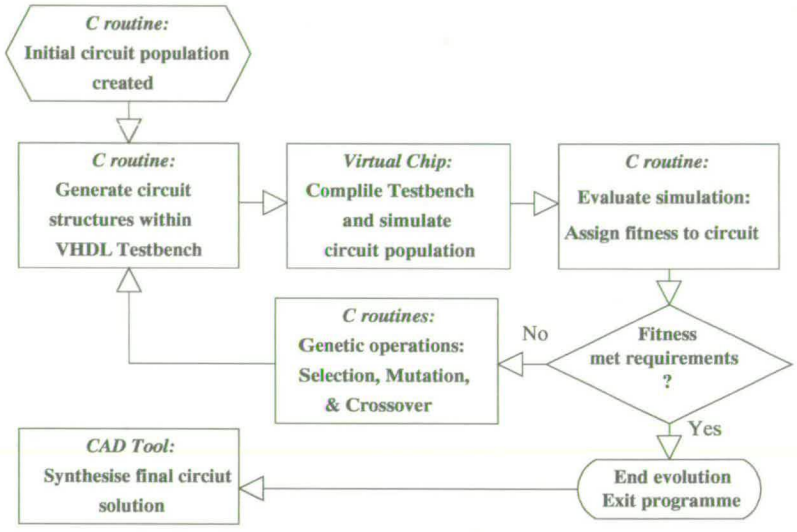


Figure 3.7: *Execution flow and coding format of the genetic algorithm and Virtual Chip evaluation environment.*

3.3 Implementation and Results

The following section highlights the results obtained when using the Virtual Chip EHW platform to autonomously generate three types of DSP circuit evolved with both timing, area and functionality constraints. The quality of circuit solution based on the performance of the genetic algorithm is further investigated through comparison of two component libraries. The first library uses both the primitive *and* functional logic elements detailed in section 3.2.1, and equates to functional-level evolution. The other component library uses only gate primitives, and reflects gate-level evolution. The following suppositions were used to provide a basis for analysis:

- The evolution of circuits using larger functional building-blocks directly translates into an improvement in the number of successful circuit solutions uncovered by a genetic algorithm
- The implementation of functional elements reduces the number of generations required to identify a correct circuit solution;
- A circuit encoded to implement functional elements is shorter in length than an equivalent circuit encoded to utilise only simple logic gates
- Circuit solutions generated using larger functional logic elements are *not* less competitive in terms of area and timing, than those generated exclusively using gate level logic elements
- The EHW platform presented provides circuit solutions of equal or better performance compared with equivalent circuits developed using standard high level CAD-based design methodologies.

Three DSP circuits were initially examined and consisted of a 2x2-bit unsigned parallel multiplier, 7-bit pattern recogniser (one's voter), and a 2-tone frequency discriminator. The multiplier circuit was chosen as it is the traditional circuit used for multiplication of FIR filter coefficients, and would provide a valuable indication as to the suitability of the Virtual Chip EHW platform for such an application. Further more, each of the three DSPs represent benchmark applications previously investigated within the field of evolvable hardware research [64, 68, 71, 78]. They also provide the foundation blocks for larger DSP applications. It should be noted that the circuits chosen represent some of the most complex arithmetic modules currently generated using the EHW design paradigm.

The performance of the Virtual Chip in generating each of the three DSP circuits, using either the functional or gate-level component libraries, is further compared with the same three DSP circuits developed using standard design methodologies and written by hand in VHDL. Using a standard HDL methodology, each of the three DSP circuits were developed in two stages. Firstly, each circuit was described at the behavioural level using VHDL, the source code of which is presented in Appendix A.1 A.2 and A.3, for the multiplier, one's voter, and 2-tone frequency discriminator respectively. Each VHDL circuit description was then passed through Cadence's *Build Gates* circuit synthesis environment [79] in order to produce a netlist tailored for a specific silicon technology. The same synthesis tool was used to synthesise each of the

circuit netlists generated by the Virtual Chip platform. The reader is referred to a more detailed account of standard digital design and synthesis methodologies using HDLs in [80].

The following terminologies are used to describe each of the three design approaches:

- *Primitive library*: Library of primitive logic elements used by Virtual Chip genetic algorithm for automated circuit design.
- *Functional library*: Library comprising primitive logic elements *and* larger macro components (see section 3.2.1) also used by the Virtual Chip genetic algorithm.
- *Behavioural HDL*: Conventional design and synthesis flow for digital circuit generation from a behavioural description written in VHDL.

Each circuit architecture was evolved ten times, and terminated after 10,000 generations if a fully correct solution (fitness of 1.0) had not been found. Evolution was halted as soon as a correct solution was discovered. All three circuits were constrained by global timing parameters to operate no slower than 10 MHz.

In order to provide a common basis for comparison, all circuits generated using either the *Primitive library*, *Functional library*, or through *Behavioural HDL* were synthesised using the same silicon technology vendor. A technology library describes the physical characteristics (such as timing and area) of the logic components associated with a specific fabrication process. Alcatel Microelectronics' *0.35μm CMOS MTC45000* technology was therefore used as the common library platform for comparison, and is the default technology library used throughout this thesis.

3.3.1 Genetic Algorithm Performance Using *Primitive* and *Functional* Component Libraries

Table 3.2 displays the averaged GA performance obtained after 10 runs for each of the circuits analysed. The results show that the *functional library* provides the genetic algorithm with a significantly better success rate when generating correct 2x2-bit multiplier and 7-bit pattern recogniser circuits. No complete solutions were found when using the *primitive library* to generate the pattern recogniser. This result is supported by findings published by Levi et.al. [71], and support the supposition that logic elements functionally more significant than gate

primitives enable the genetic algorithm to correctly generate more complex circuits.

Both the multiplier and pattern recogniser circuits generated using the *functional library* displayed a better average fitness, influenced by higher success rates. In addition, the genetic algorithm required fewer generations to find a correct solution using the *functional library* when compared to same circuits evolved using the *primitive library*. Due to the poor performance of the *primitive library* in unsuccessfully generating the pattern recogniser, this comparison could only be drawn from the 2x2-bit multiplier and 2-tone frequency discriminator circuits.

Column five in Table 3.2 provides an indication as to when, on average, the genetic algorithm finally became 'stuck' on a sub-optimal circuit and was unable to find a better solution. Such *local optima* are well known to hinder a GA search, resulting in periods of *stasis* where no improvements on a current solution are found. Results indicate that neither component library provided the genetic algorithm with consistent improvement through out every run, and in some cases stasis was reached well before forced termination at 10,000 generations.

Component Library	Success Rate	Average Fitness of Final Solution	Average Number of Generations if Successful	Average Generation Before Final Stasis	Logic Elements in Chromosome
2x2-bit Multiplier					
Primitive library	0.3	0.9750	3370.7	4082.4	30
Functional library	0.7	0.9922	2780.0	1187.0	15
7-bit Pattern Recogniser					
Primitive library	0.0	0.8930	NA	4769.6	50
Functional library	0.43	0.9794	2407.5	6726.2	15
2-Frequency Discriminator					
Primitive library	0.5	0.89480	9630.0	7812.8	100
Functional library	0.33	0.8672	7816.5	7770.5.2	30

Table 3.2: *Comparison of DSP Circuits Generated by Genetic Algorithm Using Different Logic Library Implementations.*

Figure 3.8 shows the output response of the discriminator circuit written in behavioural HDL. The circuit was designed to respond to a change in input frequency with an integration time of one complete impulse period. The input impulse frequencies were chosen to be 2.5 MHz and 833 kHz (one 4th and one 12th the operating frequency at 10 MHz). The fitness of an evolving circuit was based upon how well the circuit matched the response of the behavioural model. Thompson "evolved" a two-tone frequency discriminator successfully in [78]. Using his gate-

level approach circuit feedback was permitted. For this reason, feedback was also enabled on the Virtual Chip.

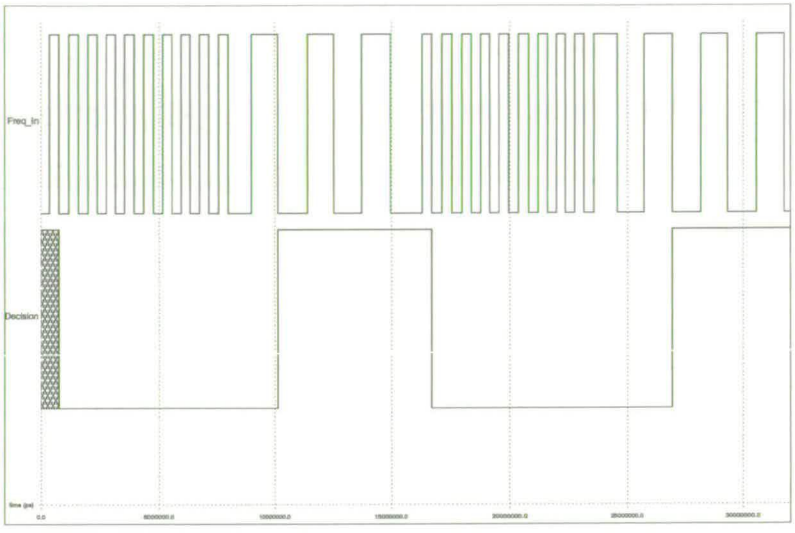


Figure 3.8: *Output response of 2-frequency discriminator from behavioural HDL model.*

Table 3.2 shows that, when using the *Functional library*, the genetic algorithm performs slightly better, on average, at finding successful solutions than when the *Primitive library* was implemented. In addition, the number of generations required by the GA to generated a correct solution is approximately 20% less using the *functional library* than that using the primitive approach.

It should be noted that the number of logic elements required to encoded each chromosome using the *functional library* was at least half that needed to encode chromosomes implementing the *Primitive library*. The number of logic elements used for each component library, and for each type of DSP circuit were empirically derived so as to obtain optimal performance and thus a fair comparison of the actual chromosome length required. However, on average, functional logic elements utilised by the GA are two to three time larger than gate primitives. The number of logic elements needed to encode chromosomes using the *Primitive library* is therefore two to three times larger than the number of logic elements employed using the *functional library*. The chromosome lengths shown in Table 3.2 reflect this.

The success of the *functional library* over the *Primitive library* can be further justified by investigating the relative size of the search space produced by each approach. Both component libraries are subject to the same fitness parameters, and both must correctly match the number

of output bits corresponding to the lookup table of each DSP circuit. However, the greater functionality of logic elements available to the *functional library* means that fewer components are required to successfully encode each circuit. This translates directly to a decrease in the search space with respect to chromosome length, and can be formalised as follows:

$$S_i = C_i^{N_i} \tag{3.1}$$

Where S_i is the search space for a given circuit architecture i, C is the number of different logic elements available to the GA from the component library, and N is the number of logic elements used to encode the circuit in a chromosome. For example, consider the search space size associated with the 7-bit pattern recogniser. In this example 50 logic elements were used to encode the pattern recogniser using the *Primitive library*. Table 3.1 shows that 14 distinct logic elements are used in the *primitive library*. The search space is therefore calculated at: $14^{50} = 2^{57}$. Alternatively, the search space for the *functional library* can be calculated at: $28^{15} = 5^{21}$. Whilst this calculation of search space size is crude, it adequately demonstrates the potentially huge differences in the magnitude of search space resulting from a functional vs gate-level approach to circuit evolution, and provides evidence that gate-level evolution restricts the complexity of circuit which can be generated because of the prohibitively large search space produced.

Figure 3.9 displays the average performance of the genetic algorithm when evolving all three circuit architectures.

3.3.2 Analysis of Timing and Area Performance

Table 3.3 displays both timing and area statistics of the three arithmetic circuits under investigation. Each circuit is identified as having been generated using either the *primitive library*, *functional library*, or behavioural HDL implementation. Timing slack is defined to be the duration for which the slowest output of the circuit remained stable before the next data pulse arrives. It should be noted that *+INF* denotes that timing constraints are well within specified limits. Circuit complexity is measured in equivalent NAND gates and represents the total physical area of the synthesised circuit using *0.35μm CMOS MTC45000* technology. The complexity measure therefore takes account of transistor area and interconnect dimensions.

Results show that on average the *primitive library* produces circuits of smaller complexity than

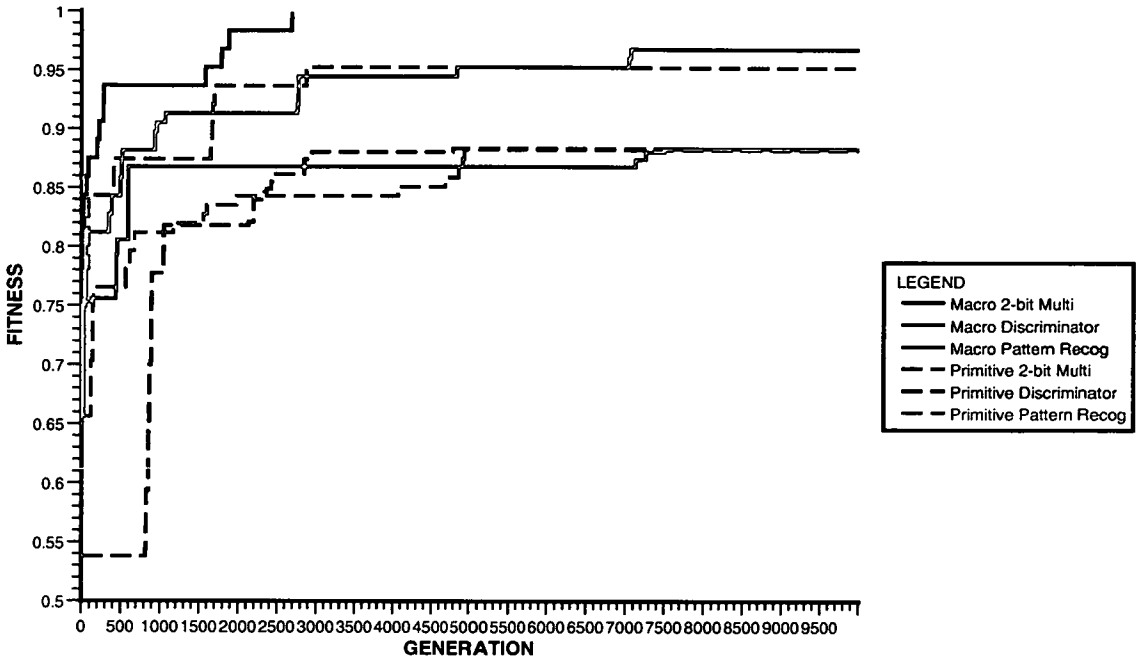


Figure 3.9: *Typical Number of Generations required by Genetic Algorithm to evolve DSP circuit structures using* primitive *and* functional *component libraries.*

both the behavioural HDL and *Functional library*. Timing is comparable in all cases. However individual solutions generated using the *functional library* are very similar to the best circuits generated using the *primitive library*, particularly for the 2x2-bit multiplier. In all cases, the best solutions generated by the genetic algorithm are either comparable or better in performance than those developed though standard behavioural HDL synthesis.

In addition to providing a technology specific circuit netlist, the synthesis procedure also provides circuit optimisation by removing redundant logic elements. This is particularly useful for circuits generated using EHW as many redundant logic elements such as through connects (Figure 3.2) will be removed. Table 3.4 presents the area and timing performance of the best solutions taken from each of the evolved circuit architectures examined. Where possible, both *primitive* and *Functional* libraries have been presented.

Results presented in Table 3.4 show marked reductions in area for both the 2x2-bit multiplier and 7-bit pattern recogniser circuits generated using the genetic algorithm. Comparison with Table 3.3 demonstrates that both circuits are between 15% and 25% smaller in area than there behavioural HDL equivalents. In all cases, the timing of circuits generated by the Virtual Chip

Implementation	Circuit Complexity in NAND Gates	Average Timing (ns)	Best Area in NAND Gates	Corresponding Best Timing (ns)
2x2-bit Multiplier				
Primitive library	10.99	93.0392	10.32	93.7459
Functional library	18.55	93.1228	10.67	94.0642
Behavioural HDL	12.68	93.5936	NA	NA
7-bit Pattern Recogniser				
Functional library	38.58	89.9210	27.33	90.8866
Behavioural HDL	20.00	91.75	NA	NA
2-Frequency Discriminator				
Primitive library	32.56	91.5315	6.67	0.93.6528
Functional library	54.59	89.8399	21.67	+INF
Behavioural HDL	75.04	+INF	NA	NA

Table 3.3: *Performance of arithmetic circuits in terms of circuit complexity and operation speed.*

Implementation	Best Area in NAND Gates	Corresponding Best Timing (ns)
2x2-bit Multiplier		
Primitive library	8.99	94.0752
Functional library	8.99	94.1303
Miller et. al.	8.66	88.9066
7-bit Pattern Recogniser		
Functional library	16.66	92.3691

Table 3.4: *Performance of GA-Based Arithmetic Circuits in Terms of Area and Operation Speed After Optimisation.*

is further improved after optimisation. Table 3.4 also draws a comparison with the 2x2-bit parallel multiplier evolved by Vasselin, Miller and Fogarty in [81]. The multiplier developed was implemented using fewer logic gates than a conventional design, a total of 7 two-input logic gates is quoted. The design presented in [81] was then converted into VHDL and synthesised using the same parameters identified above. Appendix A.4 displays the multiplier schematic and associated VHDL code. It can be seen from Table 3.4 that Miller's multiplier is comparable both in Timing and area to those evolved by the Virtual Chip. This provides a clear benchmark as to the success of developing performance driven DSP circuits using EHW over conventional design approaches.

Figure 3.10 and Figure 3.11 demonstrate the best 7-bit pattern recogniser solution obtained using the *functional library* before and after the removal of redundant logic elements. Figure 3.12 displays the 7-bit pattern recogniser analysed in Table 3.4 after full optimisation. These figures provide an indication as to the degree of cell redundancy exploited by the genetic algorithm.

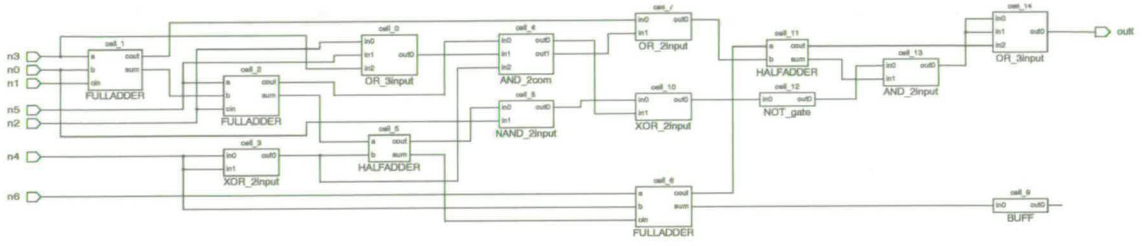


Figure 3.10: *Circuit diagram of 7-bit pattern recogniser generated by genetic algorithm using* functional library.

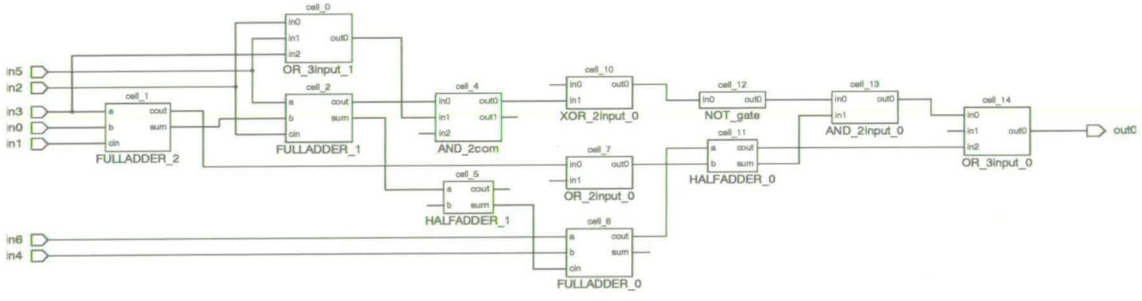


Figure 3.11: *Circuit diagram of 7-bit pattern recogniser generated by genetic algorithm using* functional library *with redundant elements removed.*

3.4 Phased Evolution in the Virtual Chip

The sheer size of the search space involved in the automated design of digital circuits can often result in the failure of an evolvable hardware framework in finding a suitable solution. Results from section 3.3.1 have shown that limiting the size of logic components available to the genetic algorithm, as with the *primitive library*, increases both the size of the search space, and the number of iterations (generations) required to find an acceptable solution. In some cases, as with the 7-bit pattern recogniser, this can prove to inhibitive.

As shown in Table 3.4, Miller et.al. have provided valuable research material from using EHW

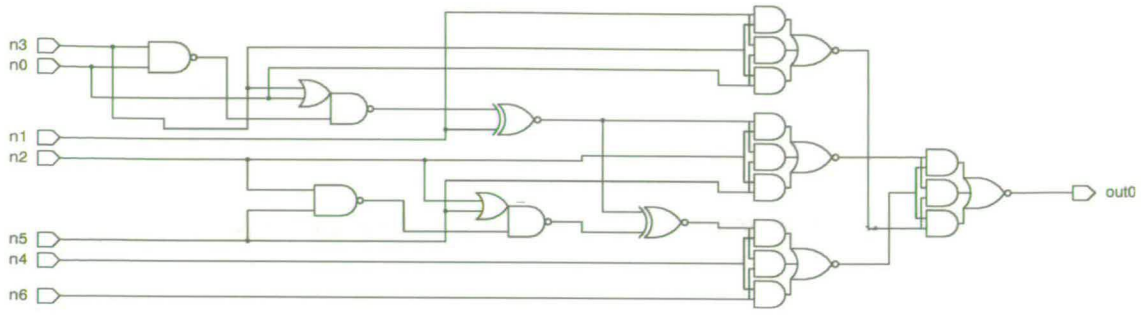


Figure 3.12: *Circuit diagram of fully optimised 7-bit pattern recogniser generated by genetic algorithm using* functional library.

and gate-level evolution to develop autonomously multiplier architectures with fewer logic components than standard multiplier designs [81]. Multiplier architectures such as those developed by Miller's EHW platform are therefore particularly relevant to high performance SoC signal processing applications, such as filter coefficient multiplication. However, the highly non linear growth in search space size and complexity demonstrated by Miller prohibits the effectiveness of EHW in generating multiplier architectures with input vectors greater than 4-bits long [12].

In addition to the number of logic elements required, and the desired circuit functionality, such complexity is well represented in the fitness evaluation of such circuits. In most cases evaluation consists of matching the output vectors of the circuit under analysis with the actual output vectors required by the desired functionality. This has been the approached adopted in this chapter for generating DSP circuits on the Virtual Chip. However, by reducing the number of possible output vectors, and thus the required complexity of a circuit, it becomes possible to develop circuits of complexity that were previously difficult, or unattainable. Figure 3.13 visualises this approach for the example of a more complex DSP circuit; a 3x3-bit parallel multiplier. If the 3x3-bit multiplier were evolved as one unit the number of correct output bits required to correctly describe the entire circuit would be:

$$B_i = 2^I * O \tag{3.2}$$

Where B_i represents the number of correctly matched output bits required for the current circuit, I is the number of inputs, and O the number of output bits required to encode the vector.

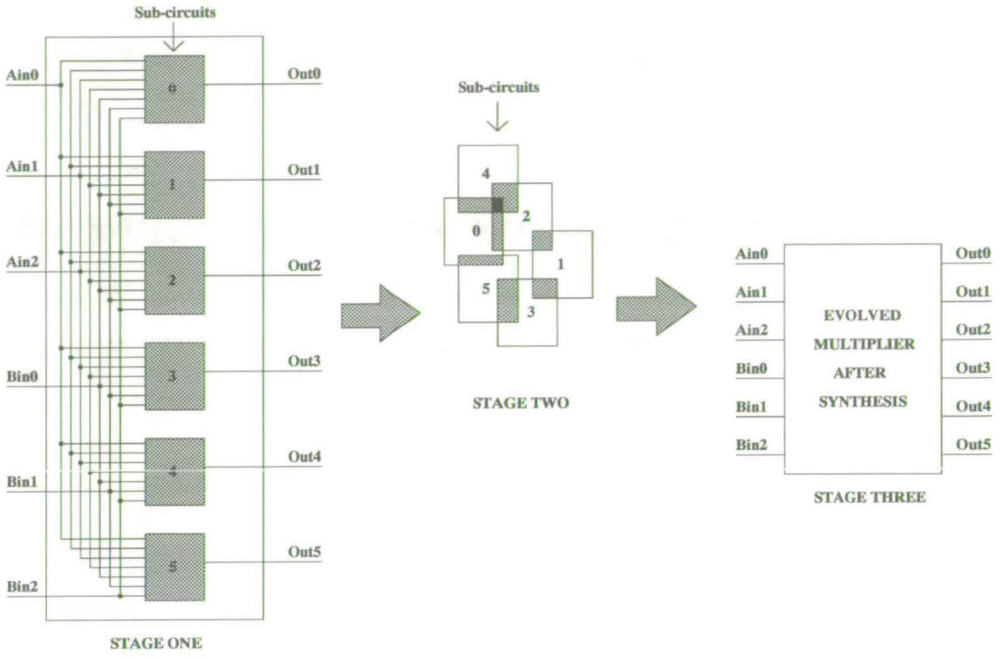


Figure 3.13: *Example of Phased Evolution For The Automated Design of a 3x3-bit Multiplier.*

However, stage one of the example circuit shown in Figure 3.13 demonstrates that, through phased evolution, the number of correct output bits required for each sub-circuit can be reduced to:

$$B_i = 2^I \tag{3.3}$$

This represents a marked difference in circuit complexity, and therefore a reduction in the size of the search space.

Stage two in Figure 3.13 denotes the removal of redundant logic between the evolved sub-circuit structures as they are combined to generate the required circuit. A benefit of evolving partitioned circuits through phased evolution is to reduce the negative effects of the high degree of epistasis, inherent in design-based EHW applications. Epistasis describes the degree of inter dependency each element in the chromosome has on the other. It has been shown that a very high degree of epistasis, as can be found in high performance digital circuits, begins to favour random search over genetic algorithm techniques [82]. It might be assumed that simply combining each sub-circuit would result in an overall circuit much larger than that developed by either a design engineer, or an alternative EHW platform. Results will show however that

the high degree of common functionality between each of the sub-circuits generated, results in large amounts of cell reuse between circuits and thus extensive minimisation is achieved during stage two optimisation.

Stage three in Figure 3.13 represents circuit synthesis enabling the designer to investigate the evolved circuit for different technologies, and confirm timing constraints are adhered to.

3.4.1 Implementation and Results

The following section details an example circuit evolved using the Virtual Chip EHW platform and phased evolution. The example presented is that of an unsigned, 3x3-bit parallel multiplier and was chosen as an incremental progression from the 2x2-bit multiplier developed in section 2. The 3x3-bit multiplier is also compared with a functionally equivalent design, generated using the same standard behavioural level HDL-to-synthesis procedure described in section 3.3, and with a 3x3 bit multiplier evolved by Miller in [12]. Both the schematic of the 3x3-bit multiplier presented in [12], and the corresponding VHDL code are shown in Appendix A.5. So as to verify reproducibility, each of the phased outputs (six sub-circuits representing each of the six circuit outputs) were evolved ten times. After evolution, specific sub-circuit solutions were chosen at random, and combined to form the final completed multiplier. The circuit was constrained to run no slower than 10 MHz, and the area of each sub-circuit was restricted by a chromosome length of 15 logic elements. A total of 90 logic elements were therefore used to encode the multiplier. However, many of these elements will be simple through-connects and many will become redundant.

The completed circuit was then synthesised to remove redundancies and calculate cell area. Table 3.5 displays timing and area information about the 3x3-bit multiplier evolved, along with the CAD-based and Miller equivalent. To further test the performance of both multiplier circuits, each was synthesised to run at 100 MHz. The results are also displayed in Table 3.5. The results indicate that, despite a slight increase in circuit complexity of 2 NAND gates, the evolved 3x3-bit multiplier operates equally as well at 100MHz as the hand designed, CAD based circuit. It should be noted that equivalent performance was obtained at this higher frequency, despite being evolved to operate at only 10MHz.

The following compares the phased evolution technique with that of the same Virtual Chip platform *without* phased evolution. Through this, the difficulty faced by single-step EHW tech-

Method of Circuit Generation	Circuit Complexity in NAND gates	Timing Slack at 10 MHz (ns)	Timing Slack at 100 MHz (ns)
Phased Evolution	45.67	+INF	1.7266 - 1.8151
Standard CAD synthesis tool	43.67	+INF	1.7395 - 1.7926
Miller et. al.	41.36	87.3771	6.3771

Table 3.5: *Comparing 3x3-bit multiplier evolved using Virtual Chip EHW platform with that of functionally equivalent circuits generated with Miller's EHW platform and by using standard digital CAD techniques.*

niques when evolving complex digital circuits becomes apparent. Figure 3.14 demonstrates the unsuccessful evolution of a 3x3-bit multiplier under the same constraints as previously detailed. In this case the total chromosome length was extended to 100 logic elements (both gate primitives and functional logic blocks), greater than the total number of elements used for the phased approach. Ten attempts were made to evolve a 3x3-bit multiplier in this way. In all cases the trend is typical of that shown in Figure 3.14, indicating that many more than 10,000 generations would be required to evolve a successful circuit. Although not substantiated, a figure of 30,000 to 40,000 generations is estimated at the current rate of progress observed.

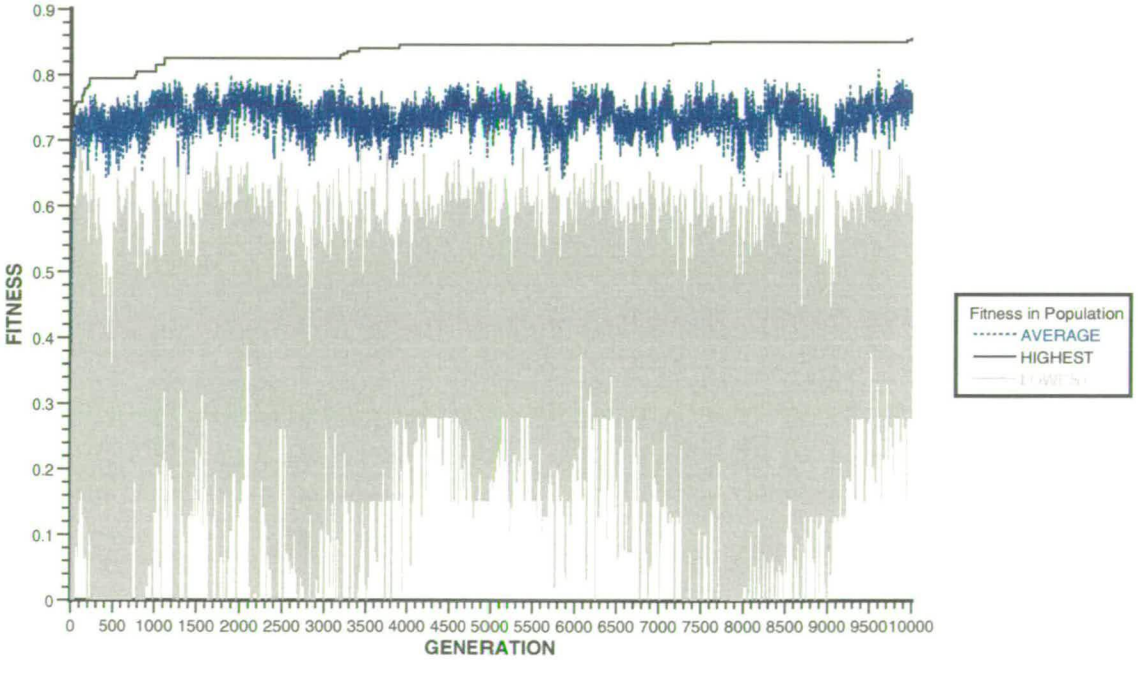


Figure 3.14: *Example of unsuccessful evolution of 3x3-bit multiplier using single-step EHW technique.*

Synthesis of Miller's evolved 3x3-bit multiplier shown in Figure 3.5 reveals that its circuit area is on average 10% smaller than that of multipliers generated through phased evolution. However, results presented by Miller et. al in [12] shown that 20 million generations were required to generate the 3x3-bit multiplier cited using a single-step evolutionary approach. Whilst is is difficult to make direct comparisons between different EHW platforms this result clearly demonstrates the difficulty in generating the multiplier circuit.

In stark contrast to the single-step approach, phased evolution provides the GA with numerous smaller complexity issues, and thus shorter evolution times. Table 3.6 displays the average number of generations taken to evolve each successful sub-circuit (a maximum of ten sub-circuits per output) for the 3x3-bit multiplier presented. It is clear that if each sub-circuit were evolved in serial approximately 20,000 generations would be required to generate the multiplier circuit. However, modern networking and efficient processors provide simple methods for executing the phased algorithm in parallel. In any case, by using phased evolution the number of generations required to evolve a 3x3-bit multiplier in either serial or parallel is considerably smaller than a standard one-step procedure, as demonstrated in Figure 3.14

Average Number of Generations to Evolve Sub-circuit					
Output0	Output1	Output2	Output3	Output4	Output5
3838	3930	7734	3508	827	55

Table 3.6: *Average Number of Generations Taken by Phased Evolution to Evolve Sub-circuits For Each Output of 3x3-bit Multiplier*

Many of the sub-circuits evolved were compared to the average number of generations taken. Table 3.6 therefore reveals a good indicator of the circuit complexity required to produce the desired output. Figure 3.15 displays an example of the best synthesised 3x3-bit multiplier evolved through the phased evolution technique. Examination of the schematic demonstrates that the most complex logic path results in the output of pin two. Figure 3.16 presents the section of digital logic relating to the sub-circuit evolved for output two after logic minimisation.

Comparison with Figure 3.15 shows that sub-circuit 2 contains the most complex logic required to achieve correct functionality. This is confirmed by the number of generations taken on average to evolve the sub-circuit.

Figure 3.17 illustrates the simplification of the sub-circuit relating to output five. The resulting circuit demonstrates the effectiveness of both logic optimisation, and the availability of

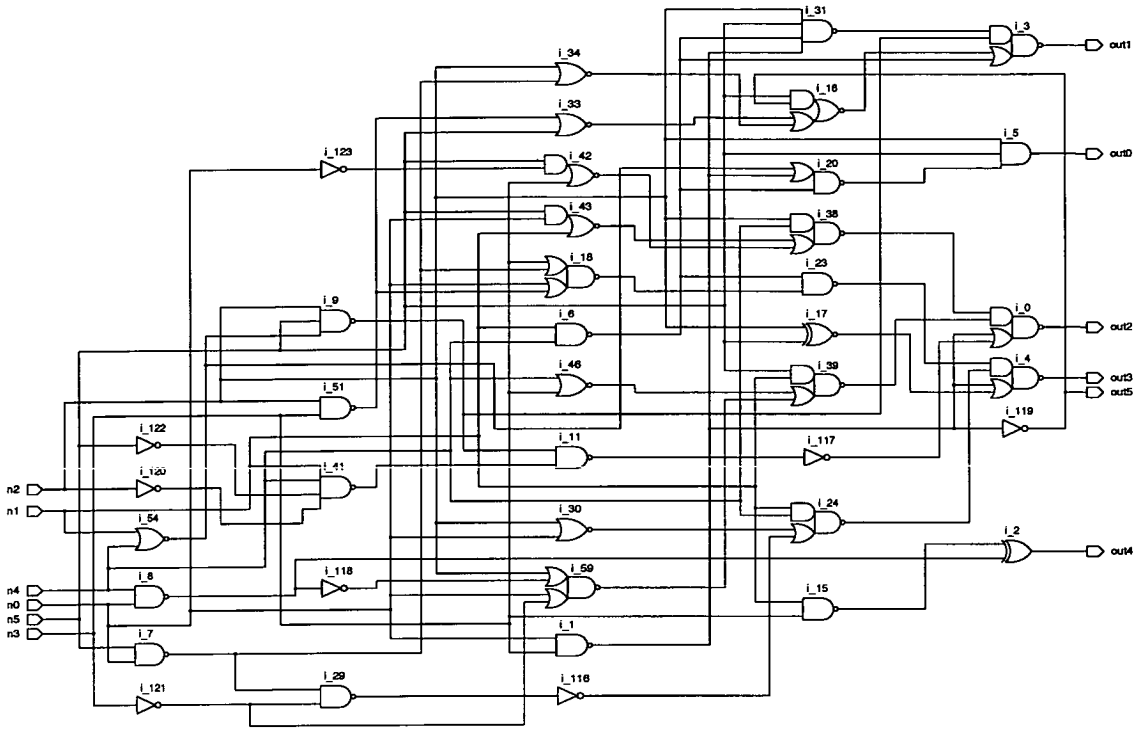


Figure 3.15: *Example of synthesised 3x3-bit multiplier generated using phased evolution technique within the Virtual Chip EHW platform.*

through-connect elements within the evolving component library (recall that each sub-circuit had a fixed length encoding of fifteen logic elements). Although not shown, the original sub-circuit representation of output five utilised a large number of through-connect and floating input elements (shown in Figure 3.2), before the removal of redundancies.

3.4.2 Limitations of Virtual Chip EHW Platform

The minimum coefficient word-length for an FIR filter is generally 8-bits. Therefore the Virtual Chip platform would be required to generate multiplier architectures considerably larger then the 3x3-bit multiplier developed using phased evolution in section 3.4. Miller et.al. [12] demonstrated the successful generation of a 4x4-bit multiplier using an array style chromosome encoding. In order to make a further comparisons with Miller's work, and to develop the complexity of DSP circuit, the automated design of a 4x4-bit multiplier was attempted using the Virtual Chip platform and phased evolution. Timing and GA parameters remained the same as those detailed in section 3.4 and circuits were generated using the *functional library*.

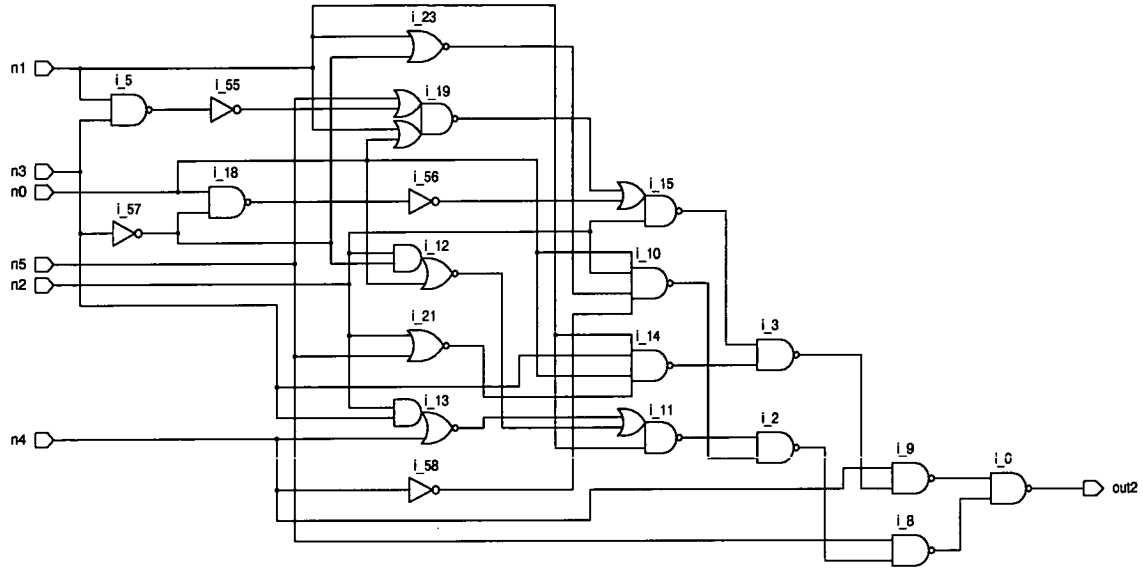


Figure 3.16: *Schematic of sub-circuit relating to functionality of output 2 of 3x3-bit multiplier.*

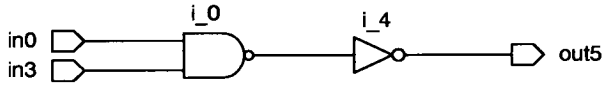


Figure 3.17: *Schematic of sub-circuit relating to functionality of output 5 of 3x3-bit multiplier.*

Table 3.7 demonstrates the results obtained for the automated design of each circuit corresponding to a single output of the 4-bit multiplier. It can be seen that only 50% of the multiplier's output circuits were evolved correctly. No successful solutions could be found for circuits which correctly described outputs 2, 3, 4 and 5. As with the 3x3-bit multiplier discussed in the previous section, a number of outputs were noticeably easier to generate using the genetic algorithm. Both the average fitness of those output solutions which were successfully generated, and the average number of generations required by the GA to produce these solutions, again provides strong indications as to non uniform distribution of circuit complexity within the multiplier search space, defined by the lookup table.

3.5 Summary

This chapter has presented an EHW platform, termed the Virtual Chip, for the automated design of performance driven DSP circuits. Within the platform a genetic algorithm was used to gen-

Multiplier Output	Success Rate	Average Fitness	Average Number of Generations if Successful	Average Generation Before Final Stasis
Output 1	0.10	0.984809	8002	3823
Output 2	0.00	0.912326	——	——
Output 3	0.00	0.836372	——	——
Output 4	0.00	0.851562	——	——
Output 5	0.00	0.842014	——	——
Output 6	0.50	0.977431	3734.4	3389.8
Output 7	1.00	1.000000	1585.7	——
Output 8	1.00	1.000000	198.7	——

Table 3.7: *Success of Virtual Chip EHW platform to generate 4-bit multiplier using phased evolution.*

erate a number of benchmark circuits based on published research, including multiplier architectures which would be required for FIR filter coefficient multiplication. All of the DSP architectures were required to operate within specified timing constraints, and were also limited in area by the number of logic elements chosen to represent each circuit. Results show that the genetic algorithm successfully generated more circuits solutions for each of the DSP applications using a library consisting of both primitive and larger functional logic elements than when only primitive logic elements were provided. The genetic algorithm also required fewer generations to find a correct solution when a functional library was used. Considerably fewer logic elements are required to describe circuits encoded using the *functional library*, resulting in chromosome lengths much shorter than equivalent chromosomes encoded using the *primitive library*. Chromosome length translates directly into search space size, and it has been shown that a reduction in chromosome length, as a result of using a functional component library, translates directly into a search space many orders of magnitude smaller than when only a primitive library is used.

Both the timing and area of circuits produced using the genetic algorithm were analysed. Findings indicate that the type of component library used by the genetic algorithm is largely independent on circuit performance, in terms of both timing and area . This can be substantiated as the GA produced circuits of comparable timing and area using either functional or primitive component libraries. Results show that circuits generated using the Virtual Chip are of comparable or better performance in terms of timing and area than those generated using behavioural HDL. By removing redundant logic elements, common in the circuit structures evolved, further

performance increases can be obtained.

In order to provide a mechanism for generating more complex DSP circuits using the Virtual Chip platform, a phased approach to circuit *evolution* was presented, and involved the segmentation of specific circuit outputs into individual circuit structures. A more complex 3x3-bit multiplier was evolved using this approach. Analysis revealed that logic element reuse between the multipliers *sub-circuits* is high, such that after the removal of redundant logic through synthesis, surface areas are comparable to a functionally equivalent 3x3-bit multiplier generated through standard HDL design techniques. This has been attributed to the high degree of common functionality between each sub-circuit. Phased evolution partitions circuit complexity and in doing so reduces the search space into smaller landscapes, related to each sub-circuit. This segmented approach therefore reduces the associated degree of epistasis inherent in the chromosomes circuit encoding; making it possible to evolve complex multiplier circuits more effectively than a standard single-step EHW approach. Results also demonstrate the non-uniformity in the complexity of the multiplier architecture related to individual output paths. However, this approach was not successful in autonomously generating a more complex 4x4-bit parallel multiplier circuit.

The failure of the Virtual Chip EHW platform and phased evolution to generate a 4x4-bit parallel multiplier casts doubts on the success of using fine-grained component libraries to generate the more complex multiplication tasks required for digital FIR filtering. Difficulties in evolving multiplier circuits larger than 4-bits using fine-grained gate primitives which constitute gate-level evolution, are also expressed by Miller et. al. in [12, 81]. Functional-level circuit evolution provides a possibility for generating more complex DSP applications. However, the size of logic elements must be considerably larger than those presented in this chapter. The next chapter therefore presents an alternative method of generating filter coefficients without explicitly using a multiplier architecture, and details the type of logic element which would be required for implementation using evolvable hardware.

Chapter 4
FIR Digital Filtering with Multiplierless Architectures

4.1 Introduction

When implementing digital signal processing (DSP) applications in hardware, great effort is made to ensure the level of performance demanded by the consumer market on such devices. Finite impulse response filters (FIRs) constitute the back-bone of most DSP applications and are therefore typically embedded alongside other processing cores which comprise the system. This is especially true when considering system-on-chip (SoC) applications. As a result considerable design resources are poured into inovative realisations of the FIR filter algorithm in hardware. The performance and portability of hardwired FIR filters are therefore of great importance. Filter performance issues in this thesis centre around speed of processing, physical area, design re-use and device reliability; all of which contribute directly to design complexity. Design re-use is becoming increasingly important to the fast development of application specific DSP devices, such that existing architectures can be ported into new applications with minimum re-design and test overhead. General purpose DSPs, such as the TMS320 series from Texas Instruments, do not provide sufficient throughput to implement high speed FIR filters, due to their single multiplier architecture. General purpose FPGAs, such as those from Xilinx [24], are suitable for implementing dedicated filter architectures. However, the general functionality of FPGAs result in complex configurable logic blocks (CLBs) which require a high degree of interconnect. This restricts circuit throughput and increases the physical area of the device.

This chapter presents the basic theory behind FIR filtering and demonstrates a number of ways in which filters can be implemented, particularly in hardware. Various multiplierless filter design methodologies and a range of hardware architectures and devices on which they can be implemented are also presented; in addition to the major building blocks required to generate multiplier-free digital filters in hardware.

4.2 FIR Filter Theory

Finite impulse response (FIR) theory is well documented and will not be covered comprehensively in this thesis. Instead, only that material relevant to the research outlined in chapter one will be presented, in order to provide a better understanding and appreciation of the research problem investigated. Detailed coverage of FIR system theory and design can be found in [83, 84].

An FIR filter can be described as a sum of N coefficients, resulting in an $N-1^{th}$ order filter given by the difference equation

$$y(\mathrm{n}) = \sum_{i=0}^{N-1} h_{(i)} x(n-i) \tag{4.1}$$

Where $h_{(i)}$ is a weight assigned to a given coefficient, and multiplied with the input sequence $x(n)$. It is these weights which describe the behaviour of the filter. By taking the z-transform of equation (4.1) an FIR system can be described by the transfer function

$$H(\mathrm{z}) = \sum_{i=0}^{N-1} h_{(i)} Z^{-i} \tag{4.2}$$

H(z) therefore describes a filter with both $N-1$ poles and zeros. Because all the poles of an FIR filter lie within the unit circle at the origin of the z-plane, an FIR system can be described as an all pole filter which is unconditionally stable. Stability is achieved because an FIR filter is a non-recursive system, as is clearly evident from equations (4.1) and (4.2). As a result the unit sample response for the FIR system is identical to the coefficients set $h_{(i)}$ such that

$$s(\mathrm{n}) = \begin{cases} h_{(n)}, & 0 \leq n \leq N-1 \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

The frequency response $H(\omega)$ of an FIR filter can be calculated from a given set of coefficient weights, $h_{(n)}$ as follows:

$$H(\omega) = \sum_{n=0}^{N-1} h_{(n)} exp(-jn\omega\Delta t) \tag{4.4}$$

The set of coefficient weights, $H_{(n)}$, is therefore calculated relative to H(ω) by integrating equation (4.4) in the frequency domain.

Each set of filter coefficients is determined by the amount of signal shaping and the frequency response of the input signal, both of which are required to achieve the desired output response. This is specified through three criteria which constrain the filter as shown in Figure 4.1. The maximum gain of the filter, represented in decibels (dBs), is determined by the stopband attenuation given as $20log_{10}(\delta_2)$, where f_2 corresponds to the edge of the stopband. Passband ripple is defined in dBs as $20log_{10}(1+\delta_1)$ and governs the amplitude of the resulting output response, $H_A(f)$, where f_1 is the passband cut-off frequency. The transition band is calculated by subtracting f_1 from f_2, and determines the steepness of the signal step between the passband and stopband.

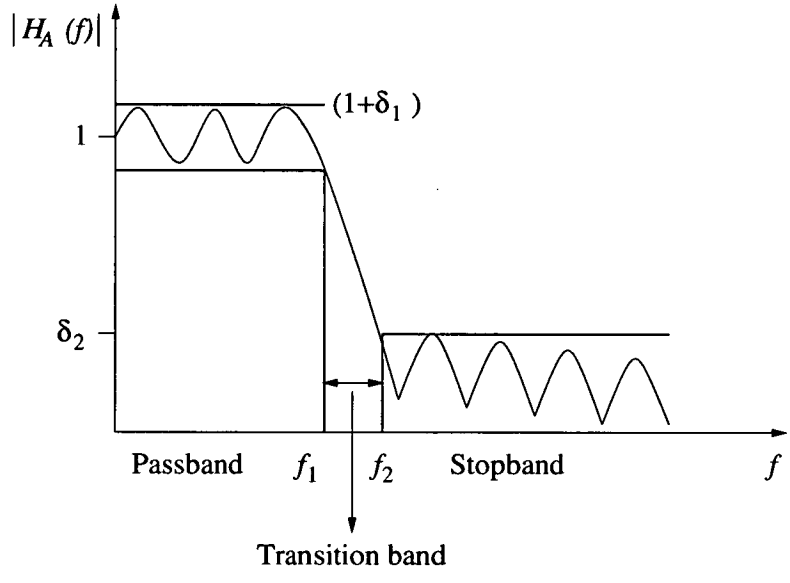


Figure 4.1: *Filter Specifications for passband ripple* $(1+\delta_1)$ *and stopband attenuation* (δ_2).

In order to obtain a finite impulse response, a window function of M samples is used to truncate the infinite time-domain sequence, c_n, into a limited range defined by M. This results in a $2M+1$ tap filter. Because only a finite number of coefficients are employed, the actual amplitude response, $H_A(\omega)$, will not match exactly with the desired amplitude response, $H_D(\omega)$. $H_A(\omega)$ is therefore calculated by convolving the desired frequency response with the frequency response of the window function, denoted as $W(\omega)$, such that

$$H_a(\omega) = H_D(\omega) * W(\omega) \tag{4.5}$$

An example of this convolution to obtain an actual frequency response is illustrated in Figure 4.2.

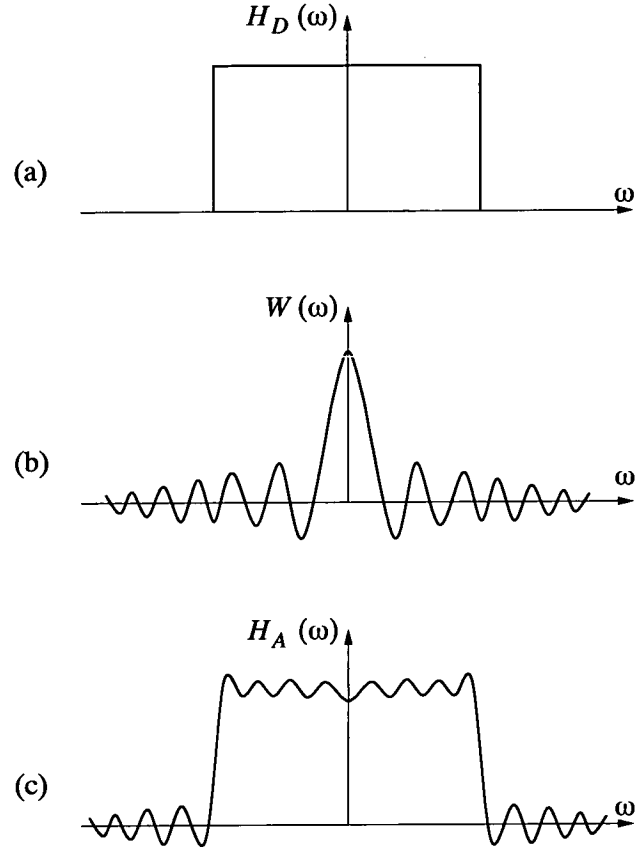


Figure 4.2: *Convolution in frequency domain for (a) desired amplitude response; (b) frequency response of input signal, (c) actual frequency response from FIR filter.*

A wide range of window functions are available, and each displays characteristics which effect the passband, stopband and transition band which together relate to the frequency response of the filter depicted in Figure 4.1. It is the job of the filter designer to determine which window function best expresses the desired frequency response for a particular signal processing application, whilst minimising the number of taps which directly translates into filter complexity.

4.2.1 Linear Phase FIR Filters

Linear phase characteristics in an FIR filter provide a means of maintaining the delay and phase relationship between frequency components of the input pulse applied to the filter, thereby minimising signal distortions. For an FIR filter to exhibit linear phase, the coefficient set must

posses conjugate-even symmetry around its centre weight. There are four ways of achieving a linear phase FIR which depend on whether the number of taps, N, is odd or even, or if the symmetry of the impulse response, c_n, is odd or even. Therefore if the impulse response displays even symmetry then $c_n = c_{-n}$, where $c_{\pm n}$ lies within the finite range $-M$ to M defined by the size of the truncation window. However, for an FIR to exhibit linear phases the filter must be made physically realisable, such that impulse responses $< c_0$ are not associated with negative time. This case is not permissible, or *non-causal*, as it infers that the FIR filter starts producing an output responses before any input stimulus is applied. In order to make the filter causal and maintain linear phase, the entire finite impulse response sequence resulting from window truncation is delayed by M samples. As a result, the impulse sample relating to $-M$ is delayed in cascade by M before coefficient multiplication. Figure 4.3 shows the effect of delaying the centre impulse response, originally located at c_0, by M, such that the impulse response sequence now lies in the range c_0 to c_{2M}.

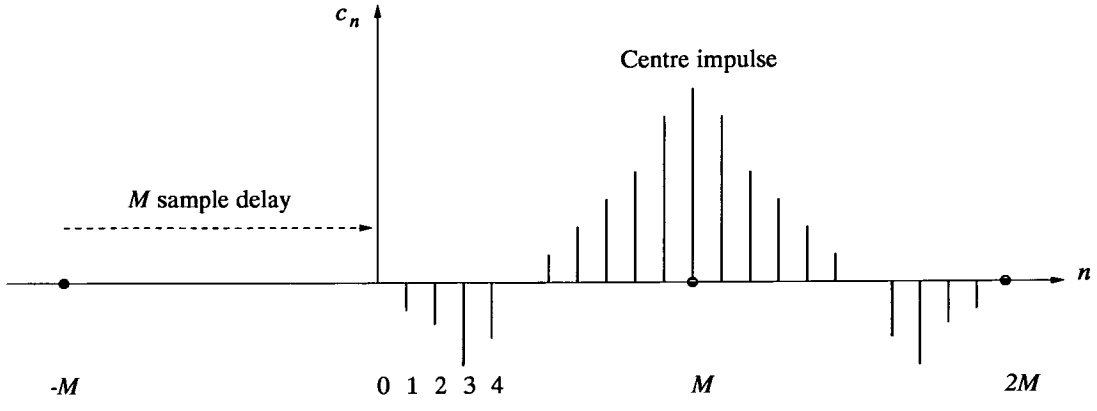


Figure 4.3: *Impulse response of causal FIR filter shifted M times.*

The transfer function of a linear phase FIR filter can therefore be written as

$$H(z) = \sum_{n=-M}^{M} c_n Z^{-(n+M)} \tag{4.6}$$

4.3 FIR Filter Implementation

This chapter focuses on the implementation of filter architectures associated with the direct form (DF) and transposed direct form (TDF) FIR systems. Both DF and TDF structures have

been chosen as they represent the most widely used form of FIR architectures. In all forms of FIR system M additions, M delays and $M + 1$ multiplications are required to implement the filter, where M is the FIR filter length (number of taps).

In an FIR filter each impulse of the input, $x(n)$, is expressed as a finite word-length of N-bits. The length of bit encoding used to represent both the filter coefficients and the input signal is important as it effects both the design of the filter response and the size and complexity of the hardware needed to implement the system. Quantising the input signal generates noise and limits the accuracy of the filter calculations. Within an FIR system word-lengths of 8 to 24-bits are usual, and depend on the signal processing application. Few FIR filters are implemented with bit precisions higher than 24-bits, as the hardware resources required filter become prohibitive. Although of interest, the effects of quantisation noise are not examined in this thesis as they are covered in depth in texts such as [83, 84].

It should be noted that many other FIR structures exist which provide architectures suited to particular DSP applications. For example the lattice FIR filter structure is extensively used in digital speech processing and in the implementation of adaptive filters. All FIR structures however require a multiplication stage in which the input $x(n)$ is weighted by M filter coefficients. It is the positioning of the multiplier elements within the FIR design flow which forms the primary difference between DF and TDF filter architectures.

4.3.1 Direct Form FIR Structure

The direct form implementation of equation 4.1 is illustrated in Figure 4.4. It can be seen that $X(n)$ is delayed in descending coefficient order from $N - 1$ before it is multiplied with the relevant coefficient and then summed.

The number of multiplications can be reduced by a factor of two if the FIR exhibits linear phase. The direct form FIR system can then be folded to produce the filter architecture displayed in Figure 4.5. N additions and delays are still required to implement the design, however, the number of multiplications is reduced from N to either $N/2$ for even symmetry or $(N - 1)/2$ for odd symmetry. This translates into a substantial reduction in hardware.

Because of the initial delay unit before each coefficient multiplication, both cases of the DF structure are particularly suited to hardware implementation using a single multiplier architecture performing a MAC (Multiply ACcumilate) operation. The MAC operation continually

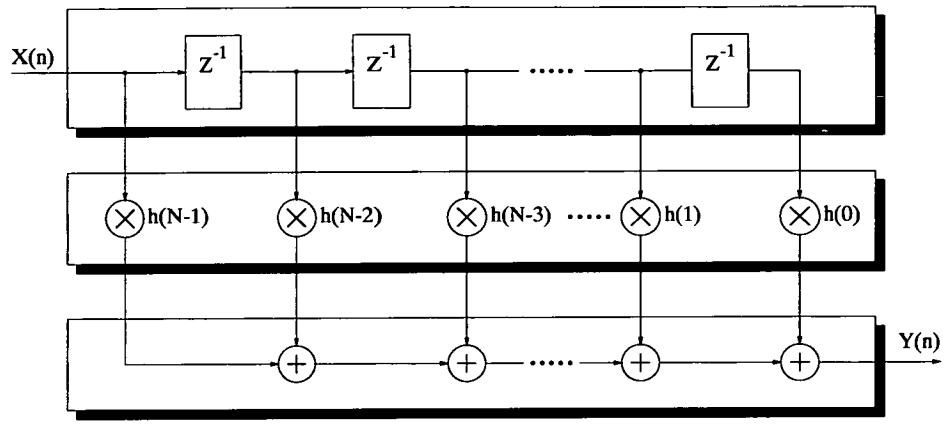


Figure 4.4: *Direct form FIR filter implementation.*

stores the result of each coefficient multiplication on every unit delay, until $X(n)$ has been passed through each filter tap. Figure 4.6 illustrates this concept. Using a single MAC in place of multiple multiplier units in either folded direct form, or direct form structures greatly reduces the area of the filter and imposes no additional delays into the system as each addition of the weighted input, $W_i(n)$, is required before data is ready at the filter output. This approach is therefore highly suited to low-power, low area DSP applications which do not require high speed data processing. Each coefficient is simply multiplexed from its corresponding memory location at the relevant time and passed to the multiplier unit. The resulting FIR filter can now be implemented using 1 multiplier, 1 storage unit (shift register) and 1 adder.

4.3.2 Transposed Direct Form FIR Structure

The transposed direct form FIR structure differs from the direct form in that the input, $x(n)$, is fed into all multiplication units simultaneously. A cascade of delays and additions then connects to each coefficient multiplier in order to impose the relevant delay. Figure 4.7 and Figure 4.8 displays the basic TDF and folded TDF structures respectively.

Both figures show that the TDF structure is capable of filtering data a factor of M faster than the DF architecture. This is because of the parallelism of the multiplier array such that the weighted calculation of $W_0(n)$ incurs no delay, and is fed directly to the filter output. However, unlike the direct form FIR a single MAC unit will incur significant latency delay if used in place of M separate multipliers, as the benefits of multiplier parallelism will be lost.

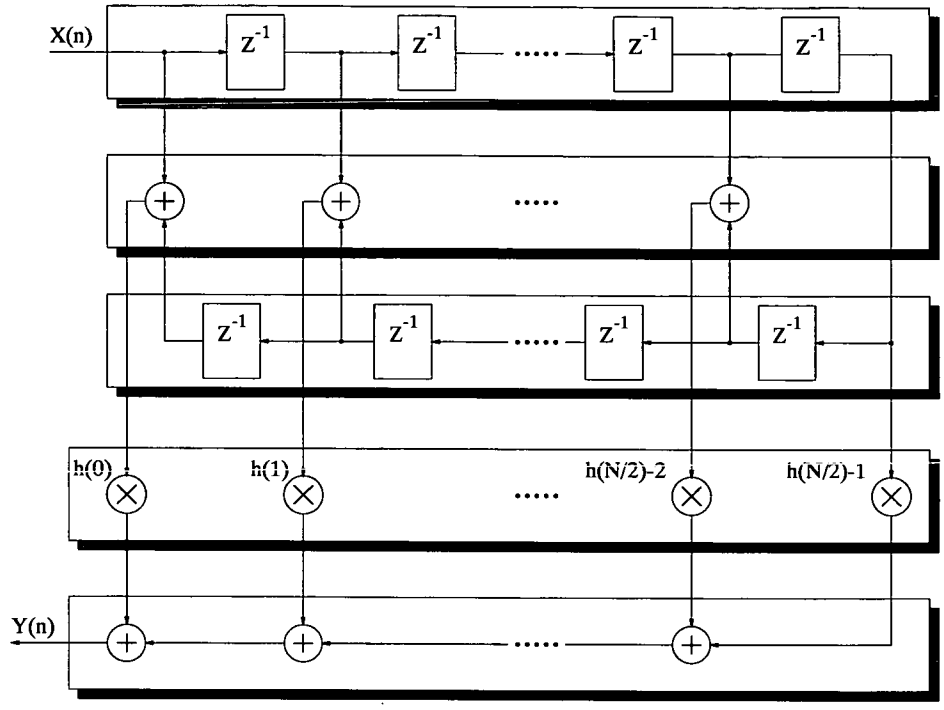


Figure 4.5: *Folded direct form FIR filter implementation (N even).*

4.4 Reduced Complexity FIR Filter Design

Within an FIR system the multiplier is the primary performance constraint when implementing either the direct form or transposed direct form structures in hardware. Multipliers are costly in terms of area, power and signal delay. Several design techniques aim to reduce FIR filter complexity and improve performance by targeting the multiplier unit.

The subset selection method relies on the design of non-uniformly spaced FIR filters [85] to produce filters requiring fewer multiplications *and* additions [86]. However this is at the expense of increased signal delay. In addition, the desired filter is not guaranteed to be of minimal complexity. Kim et.al. [87] is able to provide filters of minimal complexity by using mixed integer linear programming (MILP); hardware area is reduced accordingly. This approach is beneficial for programmable filters with time varying coefficients, where high sampling frequencies are not required.

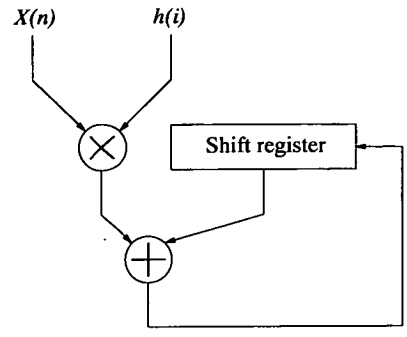


Figure 4.6: *Multiply accumulate (MAC) operator.*

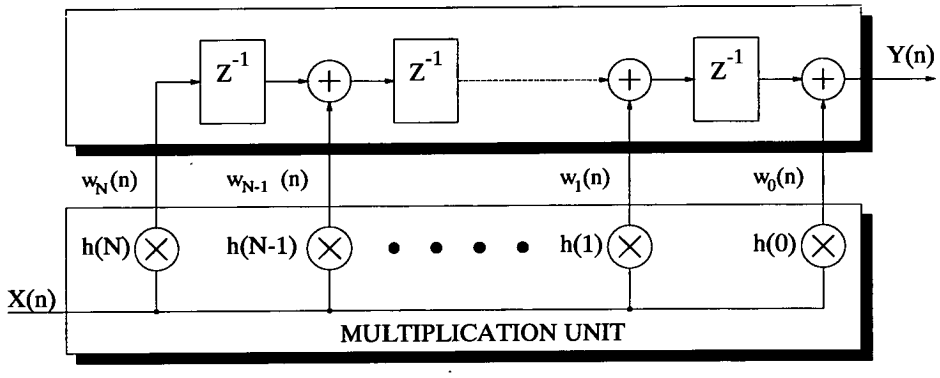


Figure 4.7: *Transposed direct form FIR filter implementation.*

4.4.1 Canonic Signed-Digit Encoding

Coefficient recoding is an effective means of reducing the circuit complexity and power consumption of fixed-coefficient filters, prevalent in high performance, application specific architectures. By fixing the coefficient of each tap, dedicated multipliers can be implemented. Dedicated multiplication replaces the multiplier unit with a series of additions, subtractions and bit-shifts which are specific to the coefficient multiplicand. Using this approach additions and subtractions become the most costly operation, and fixed bit shifts are effectively resource free. It is therefore interesting to note that the number of add operations required to realise a constant coefficient multiplication is one less than the number of nonzero bits used to encode the coefficient. The canonic signed digit (CSD) code represents coefficients in a manner which minimises the number of additions required to perform the multiplication, by reducing the number of nonzero bits in the coefficient bit-string when compared to a 2's compliment encoding [88, 89]. In order to achieve this, coefficients represented in CSD are encoded in strings of length W such that: $C = b_{W-1}, b_{W-2}...b_0$. Each b_i then takes on a value in the set $\{\bar{1}, 0, 1\}$ where '$\bar{1}$' is

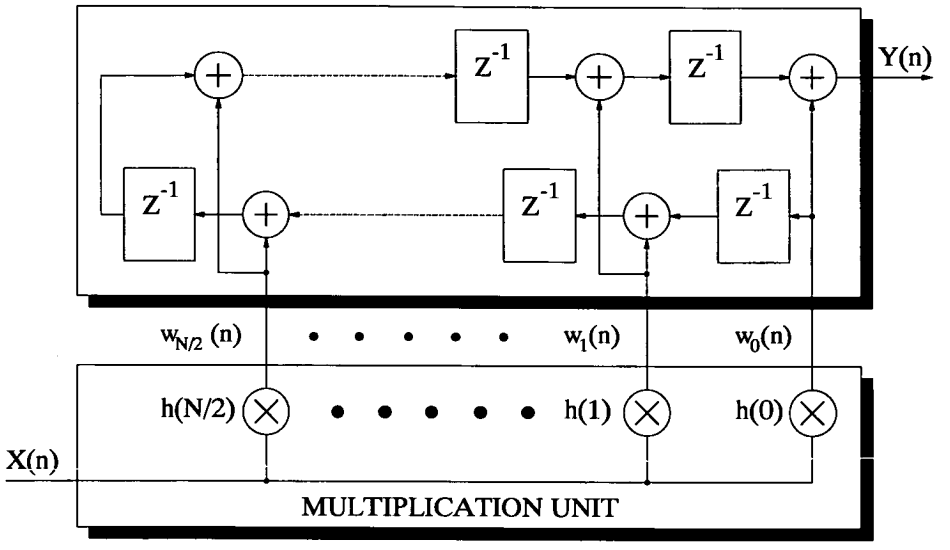


Figure 4.8: *Folded transposed direct form FIR filter implementation.*

used to denote a subtraction operation and '1' an addition. The position of the bit in the string denotes its shift value. For example consider the fixed coefficient multiplication of -894 by the filter input $x(n)$, where the coefficient is encoded in CSD

$$y(n) = 0\bar{1}0010000010 * x(n)$$

the multiplication may then be implemented as follows

$$y(n) = -(x(n) >> 1) + (x(n) >> 4) + (x(n) >> 10)$$

where $>>$ denotes a right bit-shift by an integer, i, which relates to the nonzero bits position in the string, and is equivalent to the scaling operation 2^{-i}. FIR filter performance using CSD is therefore governed by the word-length W and the number of nonzero bits in the coefficient representation, L. Within a CSD encoded word no two nonzero bits are consecutive in the string, hence then term canonic. As a result the CSD representation of each number is unique such that a number contains the minimum possible nonzero bits. On average, numbers encoded in CSD contain around 33% fewer nonzero bits than an equivalent 2's compliment encoding. This can be demonstrated by comparing the 4 CSD encoded coefficients shown in Table 4.1 with their 2's compliment equivalents.

Importantly, the value of L directly effects hardware complexity as an extra addition per tap is

CSD Encoding	Decimal Equivalent	2's Complement Encoding
$\bar{1}01010000010$	-1406	1101010000010
$0\bar{1}000\bar{1}0\bar{1}0\bar{1}0\bar{1}$	-1109	1101110101011
$0\bar{1}0010000010$	-894	1110010000010
$10000010\bar{1}000$	2072	0100000011000

Table 4.1: *Example of CSD encoded coefficients and their 2's compliment equivalent.*

required each time L is incremented. Samueli [89] has shown that one nonzero CSD digit is required for approximately each 20 dB of stopband attenuation. Techniques for optimisation of CSD coefficients include localised search, gradient-based and branch-and-bound optimisation algorithms [90–92]. However, genetic algorithms have also been employed to find the optimal set of powers-of-two based coefficients which can then implement CSD [93, 94], in addition to GAs which explicitly optimise CSD encoded coefficients [25].

4.4.2 Primitive Operator Filters

Bull et.al. introduced the concept of primitive operator filters (POFs), which utilise directed graphs to optimise filter coefficients using a combination of add, subtract and shift operations in order to generate reduced complexity filter architectures which use a standard 2's compliment coefficient encoding [2, 95]. The POF approach therefore replaces the entire coefficient multiplication unit with a highly distributed architecture, tailored for a specific set of coefficients. Bull describes the principle behind POF such that it: *"exploits the redundancy which exist in the direct-form structure. The underlying principle relies on the fact that partial products formed in any one coefficient-sample multiplication can be reused to assist in the formation of other product terms"*

Four POF algorithms were initially proposed each utilising one or more operations: add, add/sub, add/shift, add/sub/shift. Bull demonstrated that algorithms which utilised shifts produced by far the best results, typically a factor of two better than when non-shift based algorithms were employed. Filters which utilise fixed shifts are favoured because they require no logic and are therefore considerably smaller in area than either add or subtract units. An example of a multiplierless POF graph with n-bit shift and addition is presented in Figure 4.9. Note that the arbitrary coefficients used do not require additional re-coding as with the CSD approach, and that an impulse response (logic '1') must first be applied to the design to ensure that the

coefficients on each tap are correct.

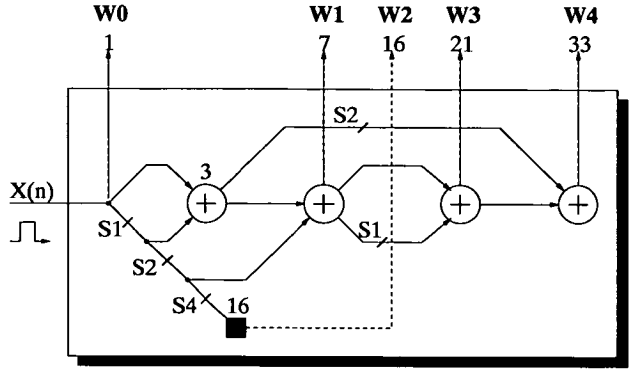


Figure 4.9: *Example Shift-add Approach.*

Arslan et.al. [96] investigated a number of configurable arithmetic macro structures designed to perform coefficient multiplication using the POF design technique. These structures were designed to be implemented on an FPGA and were shown to offer advantages both in terms of speed and physical area. One limitation of the configurable architecture proposed by Arslan lies in the single bus routing system used to connect the relevant macro structures; this bus produces a bottle-neck which restricts communication between macro elements thereby reducing throughput. In addition, the size and complexity of the configurable logic blocks (CLBs), required to implement the macro structures, severely limited the scalability of the architecture, as CLBs did not map efficiently into the corresponding filter coefficients.

POF is particularly attractive for autonomous filter design using EHW as it requires no initial encoding scheme such as CSD, and uses only three simple building blocks. Also, whilst the original work on POF utilised a heuristic search algorithm, Bull has demonstrated that POF graph synthesis problem is NP complete making it a suitable candidate for optimisation using evolutionary algorithms.

Redmill et.al presents a method of obtaining minimal coefficient sets (as with CSD) in addition to optimising filter complexity in terms of the number of subtraction and addition operations performed [97]. Redmill's approach combings a heuristic directed graph search with a genetic algorithm. Wade et.al [26] employs a similar approach to POF by providing a GA with a number of basic FIR sections such as delay elements and addition units. The GA then generates the desired filter specification using these building blocks.

Bull et.al. have shown that FIR filters designed using POF are smaller in area than those designed using the CSD approach [2, 97]. An exhaustive investigation of POF and related filter design approaches can be found in the Ph.D. thesis written by David Bull [98].

4.4.3 VLSI Implementations

Custom hardware design of the FIR filter algorithm provides greater performance than if implemented on a general-purpose DSP architecture [99]. Both ASIC's and programmable logic devices (PLD's) are used to develop custom FIR filter architectures. The choice of implementation depends greatly on the specification of the filter. Selection criteria are dependent on the filters sampling frequency, number of taps, word-length, and the need for programmable coefficients.

It has been shown that the performance of an FIR filter can be improved by replacing the multiplier with a series of bit-shifts and additions/subtractions. Physical area, and signal delay are both reduced if this approach it taken. However, multiplierless architectures rely on the accumulation of partial products, which therefore generate a fixed set of coefficients. VLSI implementations of multiplierless FIR filters range from circuits capable of sampling frequencies from 313kHz to 120 MHz, and filter orders from 32 to 64-taps [100–103]. These utilise power-of-two encodings and architectures tailored to a specific set of fixed coefficients. Coefficient programmability provides a means of extending the life of a filter core through design re-use. Whilst still tailored to the filter algorithm, these designs can be re-programmed for a range of applications; at the expense of increased complexity. Khoo et.al. presented a multiplierless filter architecture encoded using CSD and capable of implementing 32 programmable taps limited to a maximum of two nonzero CSD encoded digits[104, 105]. Woon Jin Oh et.al developed a method of reducing the length of shifters for architectures implementing programmable CSD coefficients [106]. This approach reduces area at the expense of increased computation to generate an appropriate subset within the CSD code for a specific set of coefficients.

Powell et.al. presents an investigation of the suitability of several VLSI architectures for high-speed, general purpose, programmable coefficient digital filters[107]. It was concluded that the direct form filter implementation using powers-of-two coefficients is highly effective from implementing hardware filters with 70 or fewer taps. However a generalised transversal filter architecture (GTF) may be a better compromise between hardware efficiency and ease of implementation when programmability and scalability are desired. In the GTF, tap weights are

applied to intermediate nodes which form a cascade of identical sub-filters. The output of each sub-filter is then appropriately delayed and summed to produce the desired filter response.

A number of programmable architectures have also been developed which are specifically tailored to implement FIR filters designed using EHW. Miller uses gate-level evolution comprising XOR, AND and multiplexer logic functions to generate novel filters that do not use explicit coefficients [108]. Instead filters are evolved using one of two different fitness functions. The first is based on computing the sum of the absolute differences between the actual filter response and that desired, the other is defined by examining characteristics of the Discrete Fourier Transform of the filter output. Whilst still mostly theoretical, this work demonstrates future avenues for VLSI implementations of filter applications. Flockton and Sheeham present a functional-level approach to the design analogue filters centred around a generalised building block circuit [109] which uses a number of resistors, capacitors and operational amplifiers. The architecture is demonstrated through the intrinsic evolution of a linear band-pass filter. This approach is noteworthy because of the ease in which multiple building blocks can be concatenated to realise more complex filter functions.

4.4.4 Design Adaptation and Fault Tolerance

Multiplierless FIR filter architectures have been shown to produce high performance DSP in terms of operational speed and area. The reduced complexity FIR filter design methodologies discussed above provide inovative solutions conducive for high-performance hardware architectures. Whilst a number of programmable architectures have been cited which implement these concepts, no platform yet exists in which both the filter design algorithm and the programmable architecture interact in real time. Such a platform would provide a means of online filter adaptation resulting in an architecture optimally configured for the current set of coefficients.

Device reliability is perhaps the most costly of all performance issues discussed in this chapter. It is costly as it directly impairs operational speed and increases physical area. Fault tolerant VLSI systems employ techniques such as *check-pointing* [110], *concurrent error detection* [111] and *redundancy*. Karri et.al. present a means of rapidly prototyping fault tolerant VLSI systems. Two approaches to the fault tolerant design of a 16-point FIR filter are examined. Analysis shows that designing reliability through controlled redundancy results in a VLSI design with smaller area and faster throughput than the same filter generated using a self-recovering

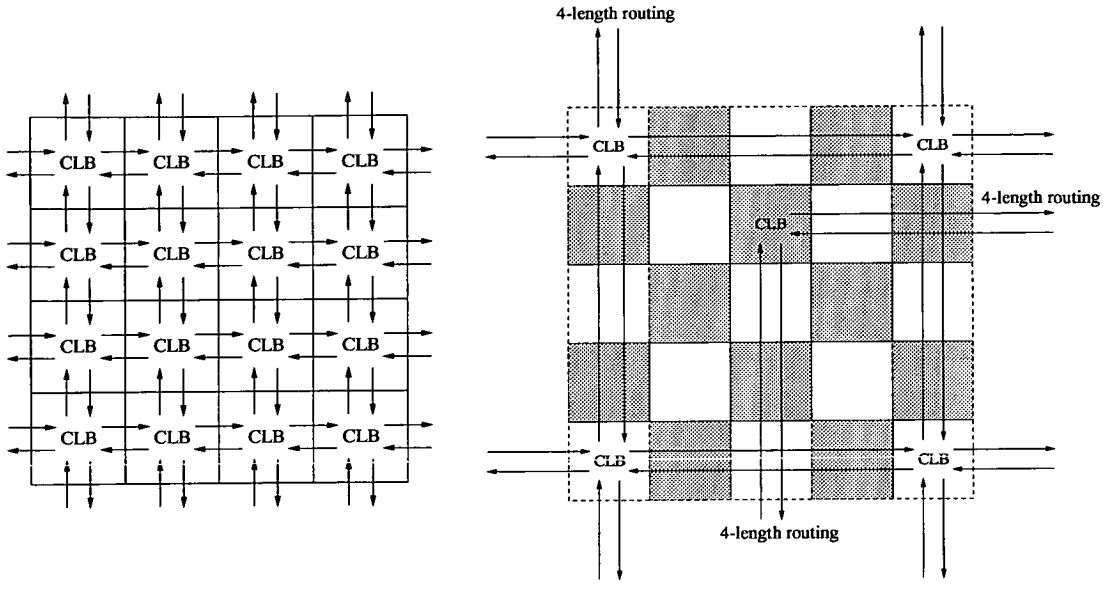
architecture [112]. The effectiveness of FIR filters developed using EHW to withstand faults is examined in detail in chapter 7.

4.5 Overview of Programmable Platforms

Programmable logic devices (PLDs) provide an alternative means of implementing DSP algorithms in hardware beyond that of more traditional approaches which use either custom VLSI hardware, generic microprocessors, or more specific DSP processors. Since the early 1990's PLD technology has branched into two distinct logic structures termed *field programmable gate arrays* (FPGAs) and *programmable logic arrays* (PLAs). Both architectures comprise an array of identical configurable logic blocks (CLBs) which are used to implement a given algorithm in hardware. A binary data string is used to configure every CLB in the array, thereby programming the PLD with the desired functionality. One of the largest differences between FPGAs and PLAs are the interconnect structures used to pass data between CLBs. Interconnect can be highly distributed as with FPGAs, or can be more restricted to rows or columns of CLBs as is the case with many PLA architectures. Figure 4.10(a) illustrates the basic interconnect topology of an FPGA. Each CLB is directly connected to each of its adjacent neighbours allowing data to be passed and received in all four directions of the array (north, south, east and west). Greater interconnectivity is further achieved by "fast" routing CLBs which are not directly adjacent. This approach can be seen in figure 4.10(b), which forms a hierarchical routing structure by grouping CLBs into 4x4 arrays. A CLB is then able to send or receive data from another CLB 4 units away, bypassing the CLBs which lie in between, which in turn frees resources. Greater levels of interconnect hierarchy can be achieved by increasing the array size of each CLB grouping.

An example of a programmable logic array structure can be seen in Figure 4.11. The architecture shown represents that of the XC9500 family of PLAs from Xilinx [24]. It can be seen that each identical function block (FB) is arranged and connected in columns, with connectivity between FBs provided using an extended interconnect matrix. PLA routing is therefore simpler than that required for an FPGA. This reduces the complexity of implementing circuits on PLAs, but also limits the flexibility of the devices when compared to the FPGA.

Both FPGA and PLA architectures implement logic functions through a combination of look-up tables (LUTs), which provide synchronous RAM, D-type flip-flops, and basic gate primit-



(a) Basic nearest neighbour FPGA interconnect structure.

(b) Example 4-length FPGA "fast" interconnect structure.

Figure 4.10: *Basic FPGA interconnect structures and CLB layout.*

ives (used for glue logic and signal multiplexing). Each of these functions is built into every CLB/FB in the array which, when suitably programmed and interconnected, produce the desired circuit functionality from the programmable device. The majority of logic functions are achieved through the LUTs. Each LUT is capable of implementing any arbitrarily defined boolean function in the form of logic truth tables, which store the bit patterns in the individual CLB/FB. As these bit patterns are stored in RAM, they can be reloaded or newly written an unlimited number of times. Circuit designs implemented on a PLD can therefore be modified and corrected by programming new bit patterns into the LUT without actually changing the hardware. More complex logic functions can be generated by spanning the truth table across of number of LUTs. Each CLBs therefore passes combinatorial bit data to the interconnect network, which can then be distributed within the array. A CLB can also store combinatorial data in D-type flip-flops which can be passed directly to the interconnect network. Multiple CLBs can therefore be configured to implement registers for storing binary words of arbitrary length.

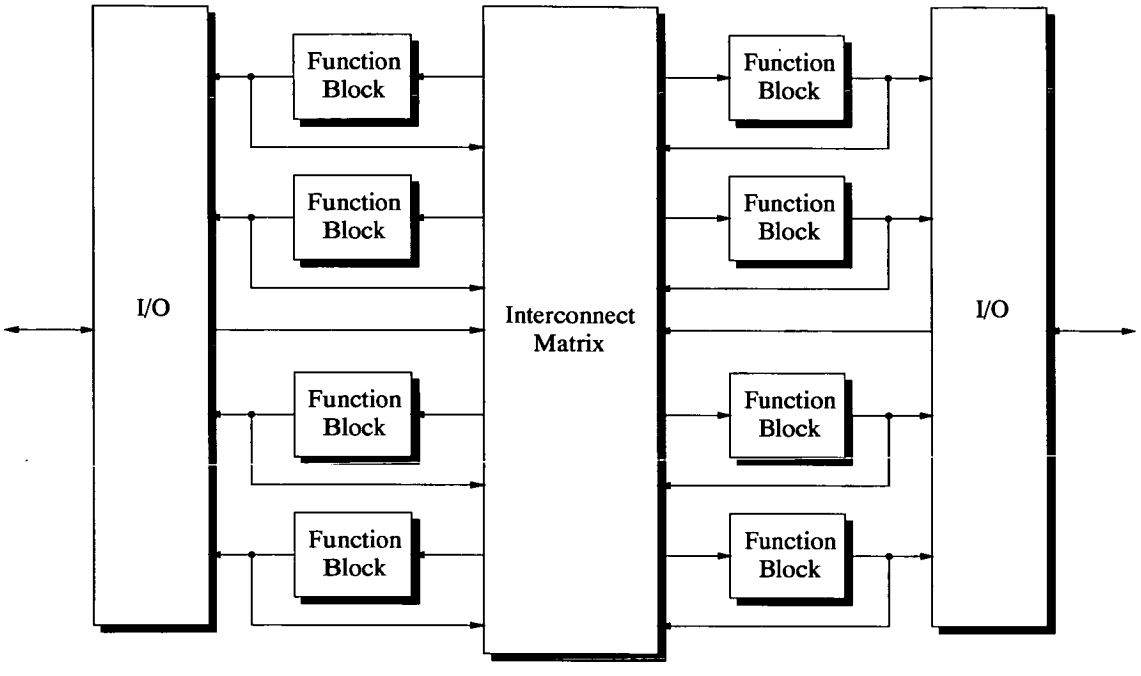


Figure 4.11: *Example of a PLA architecture from the Xilinx XC9500 series.*

4.5.1 Performing Multiplication on PLDs using Distributed Arithmetic

The majority of programmable logic devices such as those manufactured by Xilinx and Altera do not have dedicated multiplier architectures (however this is now changing). Instead multiplication is performed bit-serially across multiple CLB/FBs, an approach that requires considerable logic and interconnect resources. This limitation can be over come by exploiting the LUT-based approach to computation inherent in most PLDs, which favours a much more efficient technique for data multiplication referred to as distributed arithmetic (DA). DA is most commonly used as an efficient method of implementing the weighted sum of products, or dot product algorithm, required for applications such as FIR filtering as shown in equation (4.1). The DA approach is similar to that of POF such that one factor of each product term remains constant. Each product term therefore consists of a single input variable and a constant coefficient. Input variables are normally represented as 2's complement binary numbers such that all partial product terms are computed simultaneously in the same period that would be required to implement a single partial product. Each input string of word length N can therefore be

described as

$$X_k = -b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n} \tag{4.7}$$

Where bkn is binary data (0 or 1), $bkN - 1$ is the LSB of the data word, and $bk0$ is the signed Most Significant Bit (MSB). The result of multiplying data vector, X, of length K, with constant coefficient vector A, of length K can be written as

$$F = A_1X_1 + A_2X_2 + A_3X_3 + ...A_KX_K \tag{4.8}$$

As an example, the LUT contents for a $k = 4$ data vector can be seen in Table 4.2. A total of $2^k = 16$ possible input configurations must be referenced with the relevant partial product terms stored in the LUT. If each input word in X_K is N bits in length, then each partial product term output by the LUT must be accumulated each time the next data bit, X_{kn}, is passed. The DA multiplication function, F, therefore requires N LUT address reads, and N accumulates.

X_4	X_3	X_2	X_1	LUT content
0	0	0	0	0
0	0	0	1	A_1
0	0	1	0	A_2
0	0	1	1	$A_2 + A_1$
0	1	0	0	A_3
0	1	0	1	$A_3 + A_1$
0	1	1	0	$A_3 + A_2$
0	1	1	1	$A_3 + A_2 + A_1$
1	0	0	0	A_4
1	0	0	1	$A_4 + A_1$
1	0	1	0	$A_4 + A_2$
1	0	1	1	$A_4 + A_2 + A_1$
1	1	0	0	$A_4 + A_3$
1	1	0	1	$A_4 + A_3 + A_1$
1	1	1	0	$A_4 + A_3 + A_2$
1	1	1	1	$A_4 + A_3 + A_2 + A_1$

Table 4.2: *Contents of LUT for $K = 4$ input data vectors.*

Figure 4.12 illustrates the basic DA processor required to implement the algorithm shown in (4.2). When implemented on an FPGA the DA processor can be realised by storing all possible partial product results within a single LUT, usually spanning multiple CLBs as described

in [113]. The LUT is addressed bit serially such that each input variable is converted from an n-bit parallel word into a serial data stream which is passed to the LUT. The bit-serial input data, X_{kn}, references the LUT using the Least Significant Bit (LSB) first. Each partial product output from the LUT is then summed with the previous accumulated result and shifted one bit to the right and stored. Because all data paths in the DA processor are N bits wide, each right shift (equivalent to a divide by two) causes the LSB to be discarded. However, double precision can be retained by passing the discarded LSB via Y_{lower} onto an auxiliary shift register. This process is repeated until all the sign bits of input vector X_k are passed (simultaneously) to the LUT. Once this occurs $Sign\ control$ is read to determine the sign of the result present on Y_{upper}. If it is negative then a subtraction is performed. A minimum of N clock cycles are therefore required to process the input data vectors. Therefore, if the data width of each input word, N, is less than the number of input vectors, k, $(N < k)$ then the DA processor is in fact faster than a single parallel multiply accumulator (MAC).

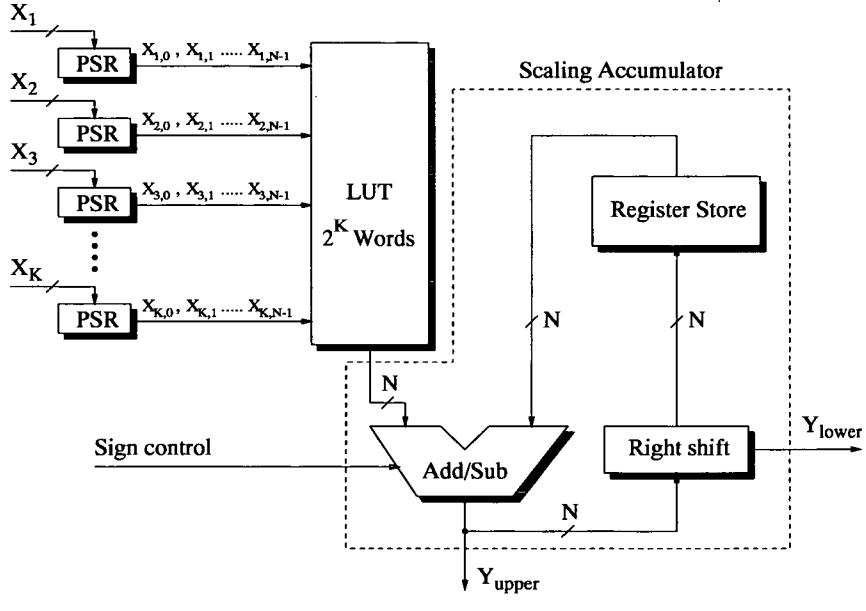


Figure 4.12: *Distributed arithmetic processor.*

An FIR filter can therefore be implemented on an FPGA using the DA technique simply by increasing the size of the LUT, thereby utilising greater RAM resources. Figure 4.13 illustrates the extension of the DA processor to digital FIR filtering.

The initial input signal $X(n)$ is firstly loaded in parallel and then converted serially to form a cascade of serial shift registers (SR) which provide the necessary tap delays and order the input

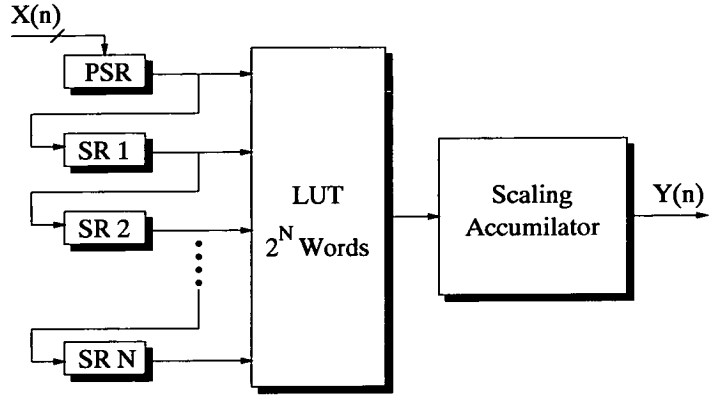


Figure 4.13: *Implementation of an N-tap FIR filter using distributed arithmetic.*

data for correct bit serial addressing of the LUT. The MAC function of the scaling accumulator then sums each partial product term output by the LUT to achieve the desired filtering function. As a result filter complexity, defined by tap length, is limited by the memory resources available to the programmable logic device.

A more detailed review of applying distributed arithmetic to DSP can be found in [114]. The article, written by Stanley White, also provides a number of techniques designed to increase the speed of DA multiplication, for example by partitioning input words into sub-words, which requires greater memory, but introduces greater parallelism into the multiply accumulate operation. Another technique aims at reducing the size of LUT required by DA. This approach is able to reduce memory resources to $\frac{1}{2}2^k$ words by using a modified 2's complement representation termed Offset Binary Coding (OBC). OBC instead casts the binary states '0' and '1' as '-1' and '1' respectively. The input X_i of word length N can therefore be used to re-write equation (4.2) as

$$-X_k = -\bar{b}_{k0} + \sum_{n=1}^{N-1} \bar{b}_{kn} 2^{-n} + 2^{-(N-1)} \tag{4.9}$$

Where $\bar{b}kn$ is the complement of the bit bkn. This approach can considerably reduce the limitations imposed by the memory resources available to the PLD, and enable the implementation of more complex FIR filters. Linear phase FIR filters can be used to further reduce the number of LUT addresses by a factor of 2. This is achieved by bit serially adding the outputs of symmetrical tap pairs, as with folded form filter implementations.

Marcos et.al presents a comparison between three classic FIR filter structures: direct form, cascade and lattice, each implemented on an ALTERA 10K50 FPGA using DA [115]. Whilst this work identified the limitations inherent in each structure, it also showed that the direct form implementation was the most scalable, and translated well in to the DA processor. An in depth overview on the implementation of transposed form FIR filters on Xilinx's latest range of *Virtex* FPGA devices can be found in [116]. An extensive overview of distributed arithmetic processors and programmable logic device architectures is presented in [117].

4.5.2 Dedicated Programmable Logic Devices

A number of recent PLD architectures have been developed which are dedicated to DSP applications, in addition to the dedicated programmable FIR filter architectures discussed earlier in this Chapter. Chen and Rabaey developed a field-programmable multiprocessor IC termed PADDI (programmable arithmetic devices for high-speed digital signal processing). The device comprises a number of identical arithmetic units connected in a similar way to the PLA architecture shown in Figure 4.11 and specifically designed for high speed signal processing applications. DSP architectures benchmarked on the PADDI include a low-pass biquadratic filter, a 3x3 linear convolver for image processing, and a RGB video matrix. The PADDI was shown to out perform a commercially available FPGA series (XC3090) produced by Xilinx at the time [118]. Rajagopalam and Sutton have recently presented an FPGA architecture dedicated to high speed flexible multiplication for demanding DSP applications [119]. Each functional block supports multiplication, addition and multiply-accumulate operations generated through a modified carry-save adder and carry logic circuitry. Again, this dedicated FPGA architecture out performs modern FPGA devices, such as those currently available from Xilinx and Altera, which must implement multiplication though LUTs, and require extensive interconnect between many fine-grained CLBs (in terms of CLB functionality) in order to configure the desired DSP implementation.

An example of a coarse-grained field programmable logic device for DSP applications is presented in [120]. The platform consists of an Arithmetic Switching Network (ASN), similar in layout to the FPGA. However, unlike the general FPGA architecture discussed earlier the ASN comprises an array of adders, subtractors and multipliers. These arithmetic operations were chosen so that the device could efficiently perform different classes of linear, non-linear, orthogonal and non-orthogonal transforms applicable to algorithms such as the Discrete Cosine

Transform (DCT), and Fast Fourier Transform (FFT).

4.6 Summary

This chapter has presented a basic overview of FIR filter theory, and underlined the relative benefits of implementing an FIR system in direct form (DF) and transposed direct form (TDF). A number of reduced complexity methodologies have been presented which target the multiplier stage of the FIR system, often replacing explicit coefficient multiplication units with a distributed series of bit-shifts, additions and subtractions. This approach is embodied by the primitive operator design methodology, which builds on logic elements used to produce previous coefficients in the current filter in order to generate the next coefficient in the set. The disadvantage of this approach is that the FIR system developed is constrained to a specific set of filter coefficients, which then binds the filter to an individual application and does not promote design reuse. Whilst this chapter has identified a number of high performance programmable architectures specifically designed to implement multiplierless filters, as well presenting more general purpose PLDs, these platforms do not integrate design algorithms such as POF which would enable the system to reconfigure adaptively. Chapter 5 therefore presents the underlying framework for an EHW platform specifically tailored for implementing programmable multiplier-free FIR filters in hardware. Evolution is to be performed at a functional level considerably higher than that used in the Virtual Chip. Instead, the POF design methodology is adopted such that the GA is able to utilise a combination of additions, subtractions and bit-shifts. The EHW platform aims to provide the flexibility of an adaptive programmable filter, with the performance benefits inherent in a fixed coefficient architecture, resulting in a high performance programmable platform dedicated for rapid prototyping of FIR filter algorithms.

Chapter 5
Developing a Programmable Framework for Filter Design using EHW

5.1 Introduction

Chapter 4 has shown that fixed coefficient multiplierless filter architectures are suitable for high performance signal processing applications. However, they are not flexible and do not promote design re-use. Whilst programmable multiplierless filters have been developed, they have not yet been integrated with design algorithms such as the POF directed graph approach which would make them adaptive.

This chapter presents the relevant building blocks identified in Chapter 4 to produce a dedicated Programmable Arithmetic Logic Unit (PALU) capable of implementing programmable multiplierless FIR filters as part of an embedded array of PALUs. Filters are to be realised within two competing programmable platforms, one inspired by the FPGA, the other by the PLA; each made up of any array of PALUs. Both programmable platforms, presented in chapter 6, are designed to accommodate the performance constraints discussed in the previous chapter which centre around processing speed, physical area, component re-use and device reliability. The genetic algorithm developed to autonomously configure the two programmable platforms for a given set of filter coefficients is also presented.

Both the PALU and genetic algorithm form the back bone of an EHW platform which has been designed to provide an adaptive, multiplierless hardware architecture, tailored to programmable FIR digital filter applications, and autonomously configured using a genetic algorithm.

5.2 Overview of EHW Platform

A further two programmable platforms are to be presented which provide a means of autonomously configuring the coefficient multiplication stage of an FIR filter using evolvable hardware.

Each of the two programmable architectures has been tailored for FIR filter design, programmability, and adaptation. Whilst the topology of each programmable platform is different, both architectures utilise the same genetic algorithm, and the same programmable arithmetic logic units (PALUs), designed to implement reduced complexity multiplierless filters. Both the PALU and the genetic algorithm have been developed in VHDL at the RTL (Register Transfer Language) level, such that global parameters can be characterised and provide a scalable logic core which can be ported into larger DSP applications. This approach can therefore be termed as *Complete Hardware Evolution*, as defined by Tufte and Haddow in [72]. The designer must then decide the data input width, data output width, and the maximum number of taps the platform can support before the EHW platform is fixed in hardware. The latter will determine the dimensions of the specific programmable platform.

The EHW platform developed is common to both the PLA and FPGA-based programmable platforms and consists of three processing units as detailed in Figure 5.1. The system controller programmes the current filter specification storing both the tap-length and coefficient variables, these are passed to the platform by the user during operation. Tap-length is passed directly to the programmable platform, whilst the coefficient variables are relayed to the genetic algorithm which must then determine how best to configure the programmable logic. The GA also verifies that the programmable platform is outputting the desired coefficients. Each tap within the programmable platform is therefore output back to the GA unit. Communication between processing units is fully synchronous.

5.2.1 Programmable Arithmetic Logic Unit

Each programmable platform is constructed from a number of identical programmable arithmetic logic units (PALU's). Unlike the more macro-based approach taken by Arslan et.al [96], a more granular structure is proposed. Each PALU is able to implement either a parallel n-bit left-shift, addition or subtraction as shown in Figure 5.2, with a bit-width dependent on the input data width of $X(n)$. Therefore as with POF and CSD approaches, explicit coefficient multiplication is removed.

A total of five control-bits are required to configure each PALU: 1-bit to determine the operation of the adder/subtractor, 1-bit for each of the routing multiplexors, and 2-bits to control the programmable shifter, which is capable of left-shifting from 0 to 3-bits. A PALU implementing a shift-by-zero acts as a through-connect. Each PALU output then feeds into a synchronous

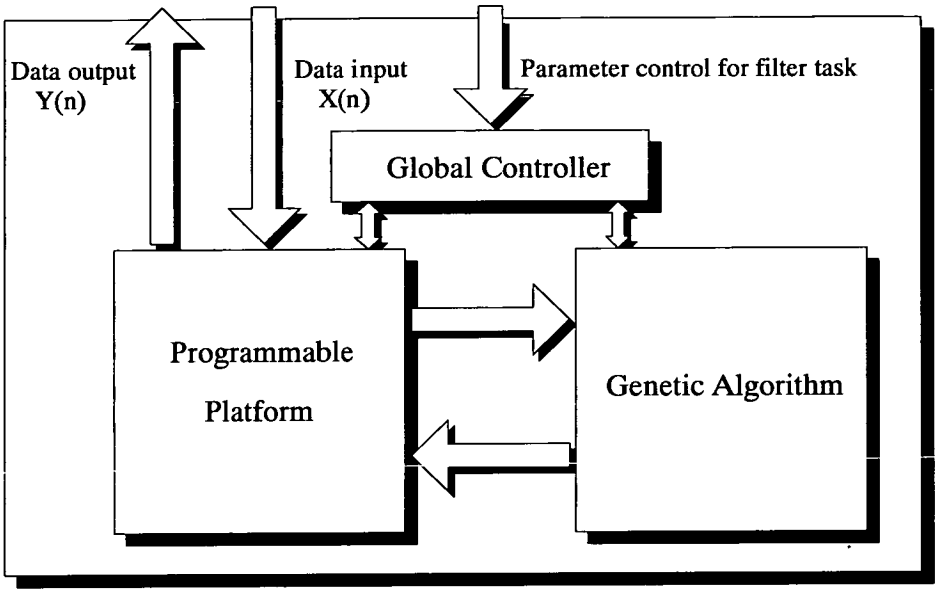


Figure 5.1: *Architectural overview of EHW platform for FIR filter implementation.*

register to create a pipelined architecture which increases data throughput.

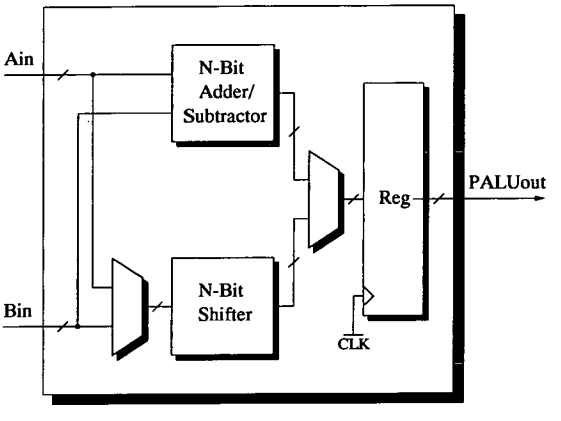


Figure 5.2: *Programmable ALU for Multiplierless FIR Filtering.*

Chapter 6 investigates a number of programmable logic topologies which utilise the PALUs illustrated in Figure 5.2. A genetic algorithm will be used to determine the most suitable topology of PALUs in which to implement a reconfigurable FIR filter using evolvable hardware.

5.3 Implementing the Genetic Algorithm

The genetic algorithm presented in this Chapter was chosen to facilitate an investigation of the main focus of research presented in this thesis, that being ***to develop the most suitable platform for implementing a high-performance digital FIR filters using EHW***. From this, the coefficient multiplication stage has been identified as the primary unit for design automation. The genetic algorithm also provides a controlled comparison on the merits of each programmable platform as it requires no specific knowledge of either the PLA or FPGA based architectures under investigation. As a result the success of each programmable platform relies solely on the ease in which the GA is able to navigate the search space and generate the desired set of filter coefficients.

Evolutionary algorithms have been used to optimise coefficient sets for multiplierless filter applications, whilst optimising a number of addition/subtraction and shift resources [26, 94, 97]. The GA presented in this chapter extends this principle to the optimal configuration of custom-built PALU's to obtain a set of desired filter coefficients within two dedicated programmable architectures. Each programmable platform is configured using a configuration string of binary data. The bit string is then used to determine the functionality of each PALU in the architecture, and the flow of data between communicating PALUs, as constrained by each platforms interconnect topology. The chromosome encoding therefore requires a binary representation like that discussed in Chapter 2. The genetic algorithm therefore differs considerably from the numeric-based chromosome encoding developed for the Virtual Chip EHW platform discussed in Chapter 3. Each programmable platform is now modified entirely by the genetic algorithm such that a population of configuration-strings are used to produce an optimal PALU configuration for a given filter specification.

A number of programmable logic devices have been used to implement EAs in hardware [17, 121–123]. These algorithms are capable of running considerably faster than those implemented on general purpose micro-processors, and are therefore suitable for applications which require online adaptation as is often required with high performance digital filters. Evolutionary algorithms are frequently mapped onto PLDs so that the fitness function can later be modified for different optimisation problems. However, faster algorithms can be achieved when implemented in dedicated VLSI hardware. Custom evolutionary algorithms implemented on ASICs can be found in [124, 125]. In such cases the fitness algorithm is fixed for a specific application. The custom ASIC approach has been implemented in this thesis so that the GA can be

embedded along side the programmable platform, making it highly suited to SoC single chip DSP devices. Figure 5.3 displays a schematic of the generic VHDL EHW platform model used to implement both the FPGA and PLA programmable architectures, and embedded genetic algorithm.

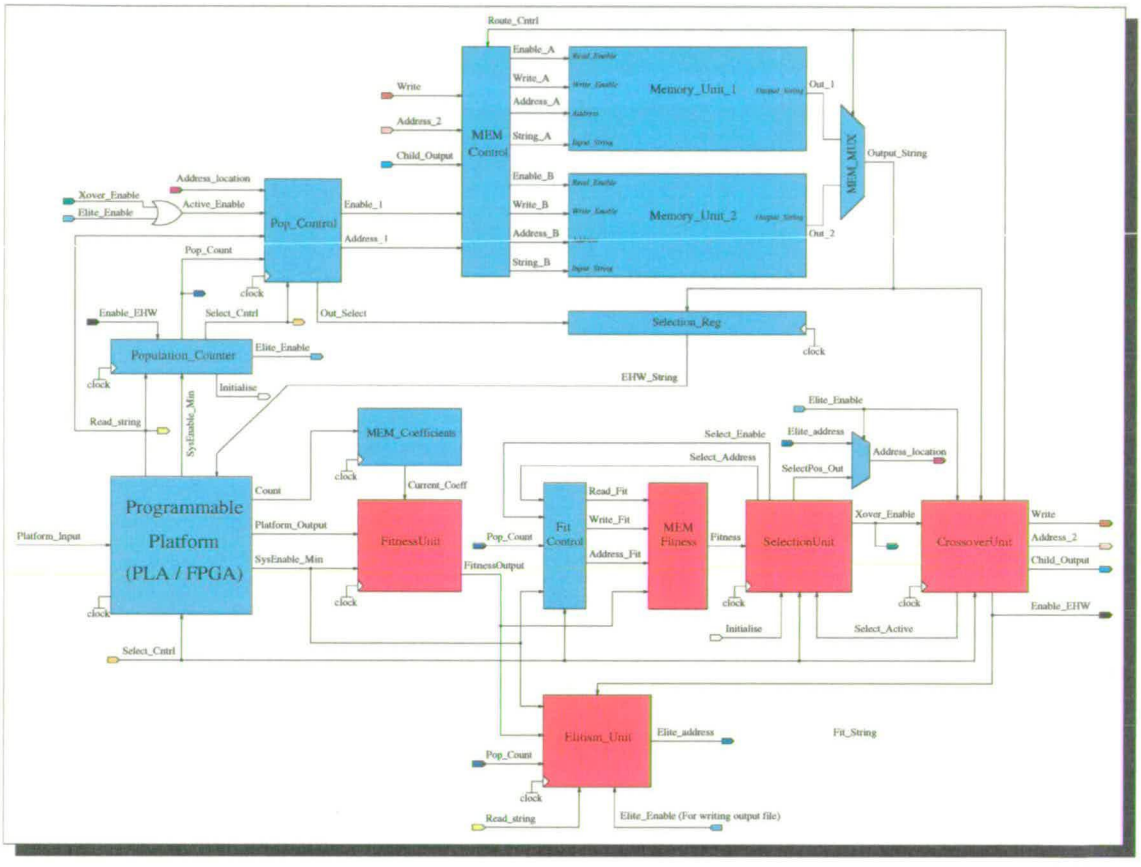


Figure 5.3: *Schematic of EHW platform including units comprising genetic algorithm and programmable platform (FPGA/PLA).*

Each unit illustrated in Figure 5.3 is described briefly below. Units highlighted in blue indicate VHDL models which may be synthesised into silicon, units in red use real valued numbers for calculation, and therefore represent high-level behavioural VHDL descriptions.

Memory_Unit 1 and 2: These memory arrays store the population of configuration strings required to program either the PLA or FPGA architectures. Each generation one memory unit is triggered to be read-only, while the other is write-only. These read and write states are determined by a logic high on the the inputs *Read_Enable* and *Write_Enable* respectively, and are present on both memory arrays. This is to enable both the fitness assessment of the parent

population, and the creation of a new offspring population. As a result the memory arrays toggle between read-only and write only modes each evaluation cycle (generation) so that once an offspring population is created it can be evaluated in the following generation. The address of each configuration string in memory is passed via the *Address* input, whilst the data itself is fed bit-serially through *Input_String*.

MEM_Control: This unit governs which *Memory_Unit* is to be read from or written to in each generation. The input signal *Route_Cntrl* determines when this transition occurs, and is only toggled when each configuration string in the current population has been evaluated and the resulting offspring strings written. Inputs *Address_1* and *Address_2* are then multiplexed between *Memory_Unit_1* and *Memory_Unit_2*, where *Address_1* selects configurations strings to be read into the programmable platform, and *Address_2* points at the relevant memory location to which the next offspring string will be written. String writing to memory is triggered by the *Write* signal, set high by the *Crossover_Unit* once an offspring string has been generated. *Child_output* then passes the new configuration string into memory. A circuit diagram of the *MEM_Control* unit is shown in Figure 5.4.

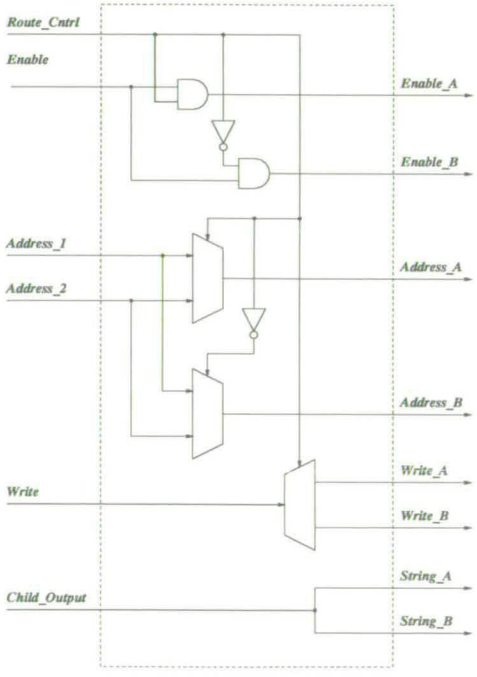


Figure 5.4: *Schematic of MEM_Control unit for memory read/write control.*

Pop_Control: Determines when a configuration string should be read from memory, and which

unit has requested the bit string. Primarily this unit simply increments the read address in memory of the next configuration string which is to program the PLA/FPGA. This is determined by the unit *Population_Counter* which passes the next address location to *Pop_Control* via *Pop_count*. Both *Selection_Unit* and *Elitism_Unit* request configuration strings via *Pop_Control* using signals *Xover_Enable* and *Elite_Enable* respectively. The corresponding memory location for both units is signalled by *Address_location*. Only when *Select_Cntrl* is low (logic '0') are both *Address_location* and *Active_Enable* recognised. When *Select_Cntrl* is high (logic '1') then the EHW platform is in evaluation mode and addressing is achieved through *Pop_Count*.

Population_Counter: While searching for an acceptable filter solution, the EHW platform has two modes of operation: evaluation mode, when configuration stings are read from memory and assigned a fitness score based on how effectively they configure the corresponding programmable platform, and evolution mode, when good solutions in the current population are selected to form offspring configuration strings which are written into memory for the next generation. These two modes are controlled by *Select_Cntrl*, which is high during evaluation mode and low during evolution mode. An internal sequential counter is used to increment *Pop_Count* so that each configuration string in the current population can be accessed and evaluated. During this period *Select_Cntrl* is held high. Once the population limit is reached, counting stops, *Select_Cntrl* toggles low, and the evolution mode begins. Evaluation mode resumes once the next generation of configuration strings is written, indicated by a single pulse from *Enable_EHW*, originating from the *Crossover_Unit*. Once this flag is received *Pop_Count* is reset and then continues to increment again.

Selection_Reg: Acts as a temporary memory store for the current configuration string programming the PLA/FPGA. The string will remain in the shift register until the performance of the platform has been evaluated. The output, *EHW_String*, is enabled by *Out_select* which ensures that the register output is delayed by one clock cycle so that memory can be safely accessed and read though *Pop_Control*. *Out_select* is therefore also governed by *Select_Cntrl* as the memory store is only required during the evaluation mode.

MEM_Coefficients: This memory unit stores the coefficient set which defines the current filter specification. Each tap output is therefore multiplexed and passed via *Platform_Output* to the *Fitness_Unit* where it is evaluated with the corresponding desired coefficient, passed to the *Fitness_Unit* via *Current_Coeff*.

Fitness_Unit: The performance of both the PLA and FPGA-based platform is assessed directly through the *Fitness_Unit*. This is achieved by determining the quality of each filter coefficient, presented to the *Fitness_Unit*, from the PLA/FPGA core via *Platform_Output*. The fitness of each coefficient is then calculated by comparing it with the desired coefficient, stored in the *MEM_Coefficients* unit. The fitness scores of each coefficient are then summed to provide an absolute fitness of the current configuration-string, formalised as follows:

$$Q\mathrm{x} = \sum_{i=1}^{T} \begin{cases} f_i/F_i & \text{if } f_i \leq F_i \\ F_i/f_i & \text{otherwise} \end{cases} \tag{5.1}$$

Where Q is the final "fitness score", T is the total number of taps, f_i is the PLA/FPGA output of the current tap and F_i is the desired current coefficient. The success of each PLA/FPGA architecture is therefore measured on the ability of the GA to successfully modify the configuration-string over a number of generations, such that a set of coefficients are obtained which most closely match those stored in *MEM_Coefficients*. One benefit of employing this comparative fitness measure was that it would be simple to implement in VHDL at the RTL level, and would translate easily into hardware. Figure 5.5 displays the corresponding circuit diagram.

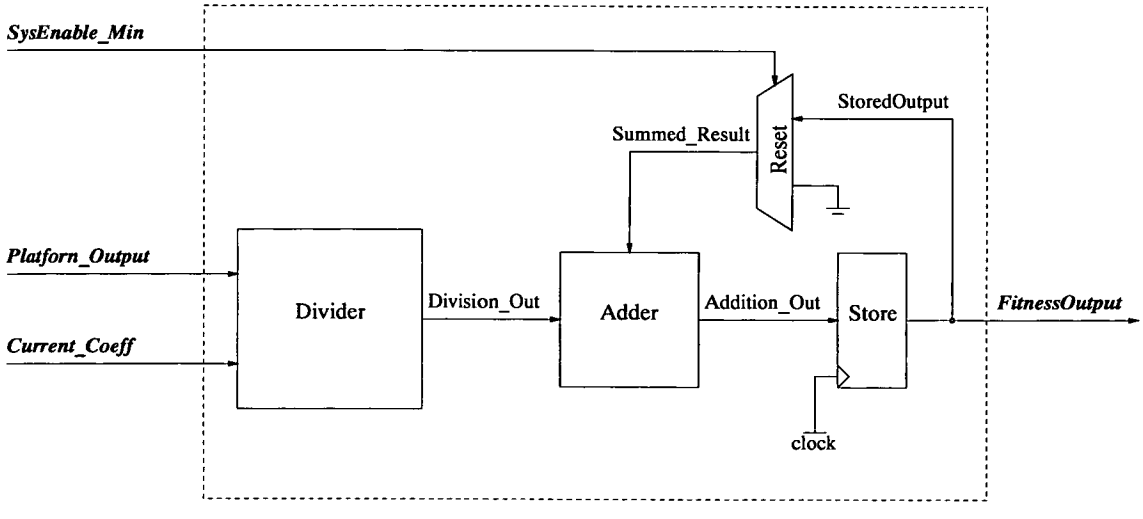


Figure 5.5: *Schematic of Fitness_Unit for calculating quality of PLA/FPGA configurations for a given set of filter coefficients.*

SysEnable_Min is used to reset the accumulated fitness score only after all the filter coefficients have been evaluated for the current string, and a new configuration string is loaded into the programmable platform.

MEM_Fitness: Is identical to *Memory_Units* 1 and 2 discussed previously except that real numbered variables are passed from *Fitness_Unit* and as a result is implemented as a separate core. *MEM_Fitness* therefore stores this accumulated fitness score, corresponding to each configuration string, in memory to be passed via *Fitness* into the *Selection_Unit* when requested.

Fit_Control: Is similar in functionality to *Pop_Control* in that it determines read/write access to memory, in this case *MEM_Fitness*. When in evaluation mode (*Select_Cntrl* = '1') memory address locations are determined by *Pop_Count* such that the position of a fitness score in *MEM_Fitness* translates directly to the position of the associated configuration string. *SysEnable_Min* acts as *Write_Enable* permitting *FitnessOutput* to be written into memory. When in evolution mode *Select_Enable* and *Select_Address* provide read access and select the desired memory location respectively. Both signals originate from *Selection_Unit* and are invoked when performance comparisons between configuration strings are made.

Selection_Unit: Two-way tournament selection was chosen as the selection algorithm for the EHW platform as it is the simplest to implement in hardware, compared to more complex algorithms such as proportionate selection, and proved successful when used with the Virtual Chip EHW platform in Chapter 3. A schematic of *Selection_Unit* can be seen in Figure 5.6

The *Selection_Unit* is first activated through *Initialise*, and thereafter via *Select_Active*. *Initialise* is transmitted from *Population_Counter* once the maximum population count is reached and *Select_Cntrl* goes low. Both these *Control* signals flag the *Random_Address_Unit* and have a period of two clock cycles. At each flagged clock cycle a randomly generated address location is passed to *MEM_Fitness* via *Select_Address*; *Select_Enable* is also set high in order to permit the memory read. This process is synchronous, therefore both *Select_Address* and *Select_Enable* activate one cycle after the *Control* flag is received. This one cycle delay between *Select_Enable* and *Control* is used to flag the remaining clock cycle when both signals are simultaneously high, so that during this period the first fitness score can be stored in *RegA* and the corresponding address stored in *Reg B*; this control process is highlighted in red. When the second fitness score is received the decision unit, highlighted in blue, compares the two scores and indicates the winner through the signal *Decision* ('0' if fitness score on *Reg A* is the greatest, otherwise '1'). The selection unit, highlighted in yellow, then passes the winning configuration string location to *SelectPos_Out*. Only once the selection has been made is *Select_flag* set high so that *Xover_Enable* can can be used to activate the *Crossover_Unit*.

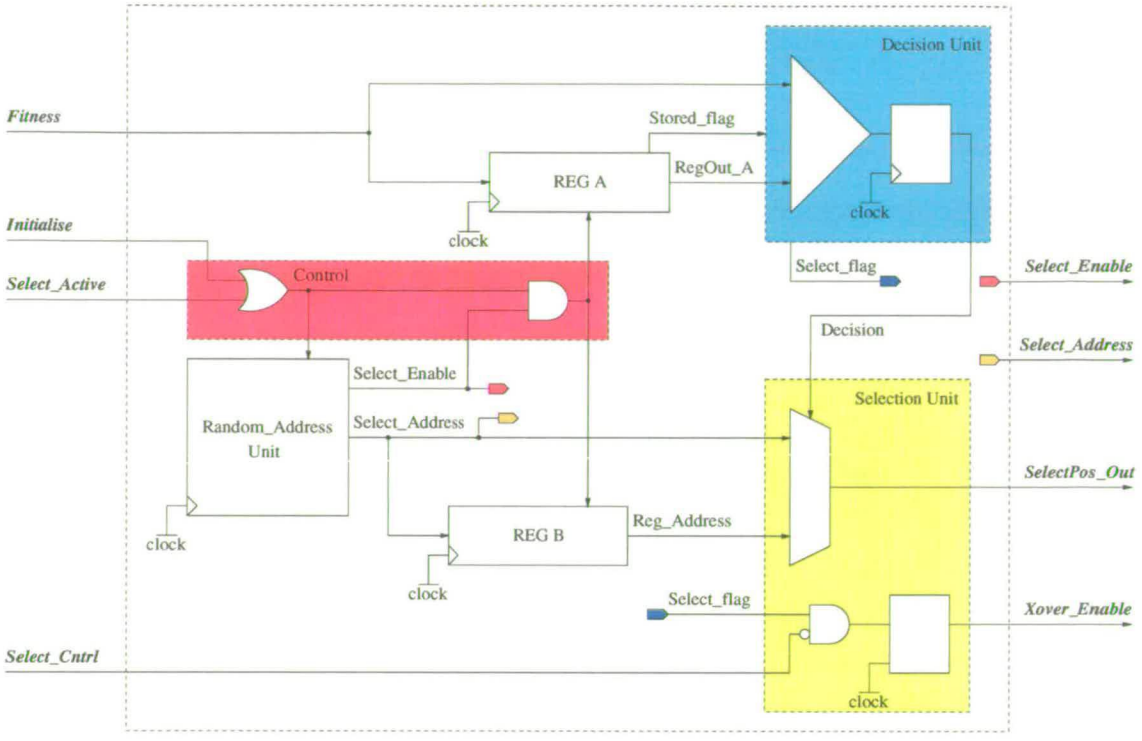


Figure 5.6: *Schematic of Selection_Unit implementing two way tournament selection.*

Elitism_Unit: Stores the address of the fittest solution in the current population, provided by the input *Pop_Count*, so that it can be re-introduced into the new offspring population unchanged. Only one elite individual is maintained each generation. On each clock event when *SysEnable_Min* goes high a new fitness score, received via *FitnessOutput*, is compared with the highest fitness score currently stored within the *Elitism_Unit*. The location of the fittest configuration string is then held in *Elite_address* until the start of the evolution mode when the *Elite_Enable* flag is set high for one clock cycle only, allowing the configuration string to be read from the current parent population and written, via the *Crossover_Unit* into the new offspring population memory.

Crossover_Unit: The primary function of the *Crossover_Unit* is to generate new offspring configuration strings through the genetic operators crossover and mutation. However, *Crossover_Unit* can also be used as means of writing parent configuration strings directly from the current memory population into the new memory population when crossover and mutation do not occur, as is done when elitism is employed. Figure 5.7 illustrates the circuit diagram of the *Crossover_Unit*.

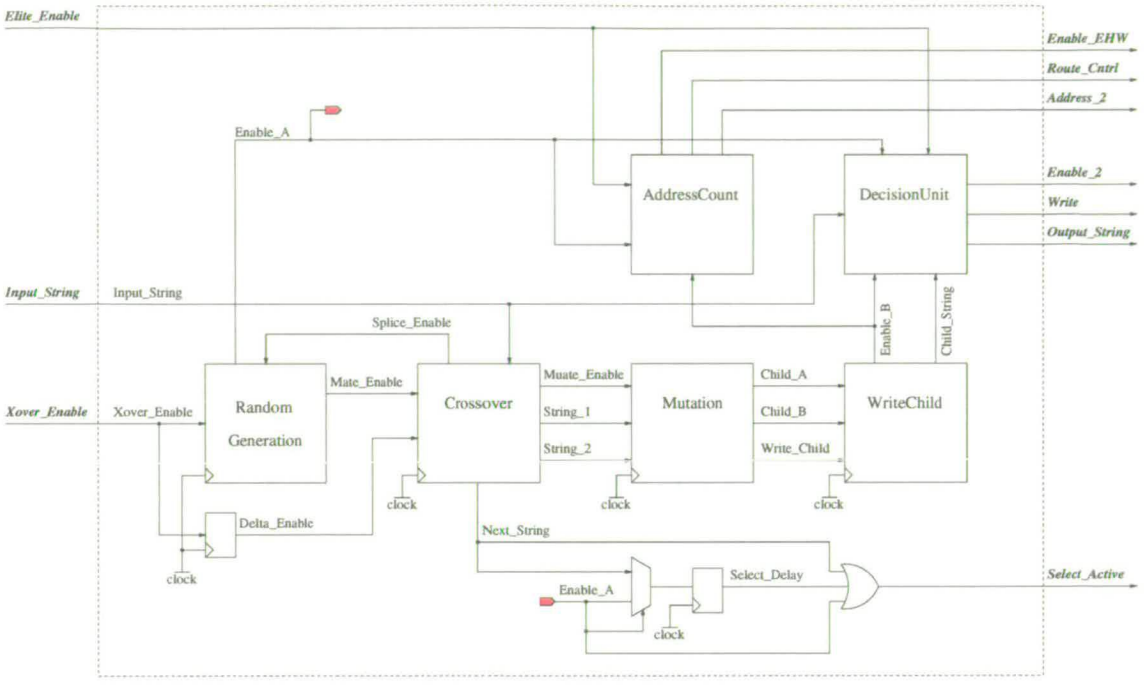


Figure 5.7: *Schematic of Crossover_Unit which implements genetic operators crossover and mutation in order to generate new offspring solutions.*

There are therefore two situations when a parent solution may be copied directly into the offspring population. The first is due to elitism, and the second is when the probability of crossover for a given string is not sufficient.

The request for an elite string read/write is issued by the *Elitism_Unit* and passed into the *Crossover_Unit* via the *Elite_Enable* flag. This flag is passed to two internal modules; *AddressCount* and *Decision_Unit*. *AddressCount* serves to control which *Memory_Unit* acts as the parent population (read-only), and which stores the offspring population (write-only) in each generation. An internal counter is used to increment the address location of the current *Memory_Unit* in write mode. A count is triggered by either *Enable_A*, *Enable_B* or *Elite_Enable*. Once the counter reaches the total population size, then enough configuration strings have been written to memory and the *Enable_EHW* flag is set high to switch the EHW platform into evaluation mode. At the same time *Route_Cntrl* is toggled, setting the *Memory_Unit* containing the new offspring configuration strings into read-only mode, and allowing the old memory population to be over written by switching it to write mode.

The *Decision_Unit* receives a copy of both the original parent string and potentially an asso-

ciated offspring string (if crossover and mutation occurred). With the *Elite_Enable* set, the parent configuration string is output directly to *Output_String*, which then feeds into the current offspring memory via *MEM_Control. Write* is also set high to enable the memory write.

Each crossover operation, activated by the *Xover_Enable* flag, is determined randomly via the *Crossover_Unit*'s internal *Random Generation* module. A user defined crossover probability is used to determine if crossover occurs, where the random number generator is bound in the range 0 to 100 (representing a 0 to 100% chance of crossover). If no crossover occurs then the *Enable_A* flag is set, acting in the same manner as the *Elite_Enable* flag discussed earlier. If the crossover probability is met then *Mate_Enable* is used to activate the *Crossover* module, which performs one point crossover at a randomly selected locus along the bit string.

Because two parent strings are required to generate offspring, the *Crossover_Unit* must then wait for a second configuration string to be passed to it from the *SelectionUnit*. The request is made via *Next_String* and extended by an additional clock cycle to generate the *Select_Active* signal expected by the *SelectionUnit*. The *Crossover_Unit*'s wait state is signalled by *Splice_Enable* which causes the *Random Generation* module to bypass probability selection, enabling the next parent string to pass directly to the internal *Crossover* module. Once both offspring strings are generated *Mutate_Enable* is set to activate the unit *Mutation* module, which applies bit-flip mutation to each offspring string with uniform probability determined by the user. Each offspring is then output in turn via *WriteChild* and passed to the *Decision_Unit*. *Enable_B* is set high to ensure that the offspring strings and not the parent are written into memory, and that the address location is incremented.

In summary, the genetic algorithm embedded within the EHW platform is parameterised as follows:

- (μ, λ) generational genetic algorithm
- Population size 100,
- User defined crossover and mutation rate
- Two way tournament selection
- One elite solution maintained each generation.

5.3.1 Analysis of Genetic Algorithm

The completed GA was simulated using Cadence's *Leapfrog* VHDL simulation environment with a crossover rate of 60% and a mutation rate equal to $1/L$, where L is the bit length. A population of four arbitrary configuration strings of 15-bits were stored in the GAs *Memory_Unit*, and each was assigned an imaginary fitness corresponding to how well the string might have configured an array of PALUs for a given filter specification. It is clear that considerably longer bits strings would be required to actually configure an array of PALUs, however, such short string lengths were chosen as it would be easy to note the effects of crossover and mutation. Figure 5.8 presents the resulting simulation waveforms relating to the GAs *Crossover_Unit*. The *Crossover_Unit* reflects the most complex aspect of the GA architecture, and adequately demonstrates the global operation of the embedded genetic algorithm.

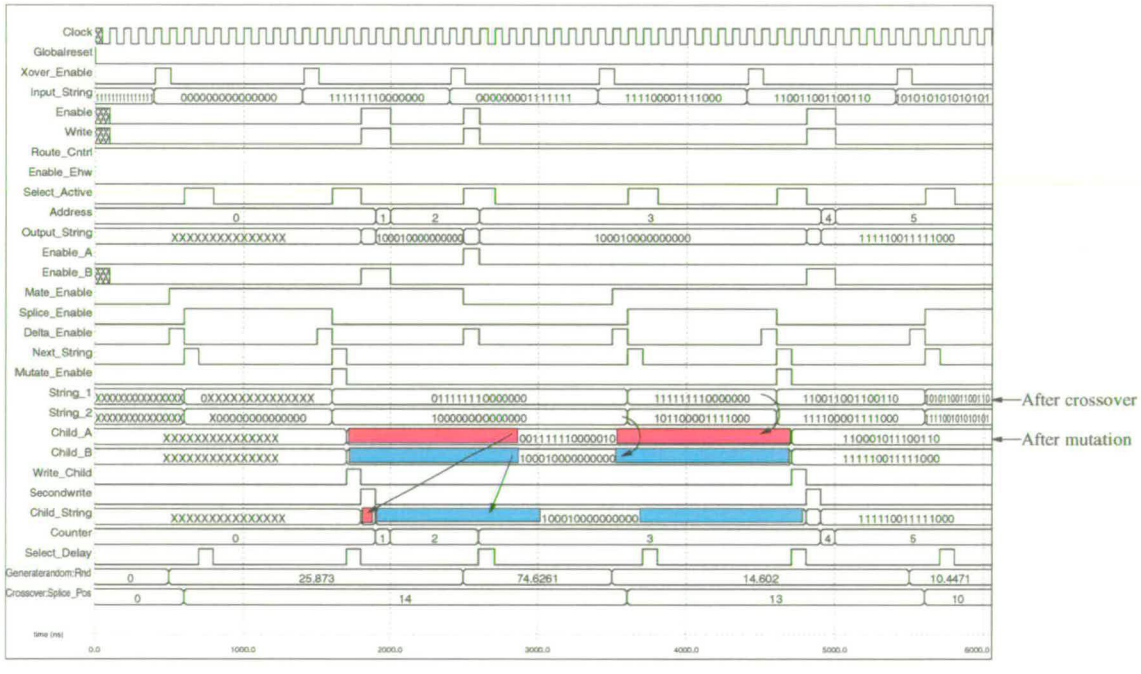


Figure 5.8: *Overview of waveform produced by genetic algorithm in EHW platform.*

Figure 5.8 clearly shows the activation of the *Crossover_Unit* through the signal *Xover_Enable* produced by the *Selection_Unit*, and the writing of *Output_String* in to memory when both *Enable* and *Write* are simultaneously high. The signal *Address* can also be seen to increment its write location in memory each time an offspring string (*Child_String*) is available for writing. The two initial configuration strings "000000000000000" and "111111110000000" present on

Input_String are shown by *Splice_Pos* (an internal signal in the *Crossover* module) to crossover at bit 14, creating offsprings "01111110000000" and "100000000000000" on *String_1* and *String_2* respectively. Mutation is then shown to occur with the correct probability on *Child_A*, highlighted in red, and *Child_B*, highlighted in blue, before being passed to *Child_String* and written to memory.

Further evidence that the genetic algorithm functions correctly is presented in detail in Chapter 6, through the successful *evolution* of digital FIR filters using the EHW platform developed in this chapter.

5.4 Summary

This chapter has presented the development of a programmable arithmetic logic unit (PALU) which constitutes the basic building block for an EHW platform developed to autonomously implement FIR coefficient multiplication. The PALU developed is designed to replace explicit coefficient multiplication with a distributed series of bit-shifts, additions and subtractions. An array of PALUs comprise a programmable platform which is then autonomously configured using a genetic algorithm with a fitness function designed to reflect a given filter specification. The GA is also employed to investigate the most suitable programmable platform for implementing high-performance multiplierless digital filters. Two of the key genetic operator: crossover and mutation (in addition to population size) can be parameterised in order to optimise the GA for the filter application.

The basic EHW framework identified in this chapter therefore comprises the GA and the FPGA or PLA-based programmable platform, both of which are written in VHDL. A detailed overview of communication between the GA and the programmable platform has also been presented, and the GA has been shown through VHDL simulation to operate correctly. Chapter 6 details an investigation into the most suitable programmable platform for digital FIR filter coefficient multiplication using EHW.

Chapter 6
Reconfigurable platforms for FIR filter implementation using EHW

6.1 Introduction

This chapter presents two programmable platforms, specifically designed to implement multiplier-free coefficient multiplication for high performance, digital FIR filter applications. The first programmable platform is inspired from a class of logic devices termed field programmable gate arrays (FPGAs), the second is from a similar family of devices termed programmable logic arrays (PLAs). Both programmable platforms will utilise the PALU detailed in section 5.2.1 of chapter 5 for the automated design of digital FIR filters using evolvable hardware.

The genetic algorithm developed in chapter 5 is used to examine a number of performance criteria which focus on the following: the success of each EHW platform in generating a specified coefficient set, the number of PALU components utilised in each array, the degree of component re-use required to produce new coefficient terms, and the ratio of left-shift, addition and subtraction operations required to implement the filter. Both the PLA and FPGA-based platforms are examined with a range of filter input, tap output and PALU interconnect topologies in order to determined the most suitable programmable multiplierless architecture.

Both the FPGA and PLA-based EHW platforms were implemented using a hardware description language (HDL) at the RTL level so as to provide accurate hardware modelling of each system. VHDL was chosen as it provided a simple means of creating arrays of PALU using the *GENERATE* statement, a feature which does not exist in Verilog. The relevant circuit layout of each programmable platform is also presented. Finally, the most successful programmable platform based on each of the performance criteria discussed is identified and selected for translation into a synthesised hardware model.

6.2 Benchmark Filter Design

In order to investigate the suitability of each programmable input, output and interconnect topology for automated filter design using EHW, a 31-tap low pass filter was selected to provide the benchmark with which both the FPGA and PLA-based architectures will be compared. The filter was taken from the industrial design of low-power filter cores for hearing aids, developed in joint collaboration with Bernafon LTD and the university of Edinburgh detailed in[126]. The corresponding coefficient set shown in Table 6.1 is highly challenging as it exhibits a large dynamic range with coefficient multiplicands scaling the filter input from 2^7 to 2^{14}, using word lengths of only 16-bits. In addition the low-pass filters gain must be no less than -52 dB. All

Coefficient Taps	Dec
W_{-15}, W_{15}	-59
W_{-13}, W_{13}	96
W_{-11}, W_{11}	-220
W_{-9}, W_{9}	461
W_{-7}, W_{7}	-876
W_{-5}, W_{5}	1606
W_{-3}, W_{3}	-3171
W_{-1}, W_{1}	10326
W_{0}	16384

Table 6.1: *Non-zero coefficients required for response of 31-tap low-pass filter.*

other coefficients are zero. Therefore 9 distinct taps are required for a folded form implementation, using the approach detailed in section 4.3.2. The corresponding filter response is shown by the blue line in Figure 6.1.

The filters transfer function was achieved by quantising the input impulse, $X(n)$, and coefficient word lengths to 16-bits. Because a number of negative coefficients are used, a 2's compliment encoding is required. Each programmable platform must therefore be characterised to accommodate these specifications. This was achieved during RTL level parameterisation of both the PLA and FPGA VHDL models.

6.2.1 Experimental Setup

Tests on both EHW platforms have therefore focused on the automated configuration of the 31-tap low-pass filter identified above. Each PLA and FPGA topology is investigated 10 times

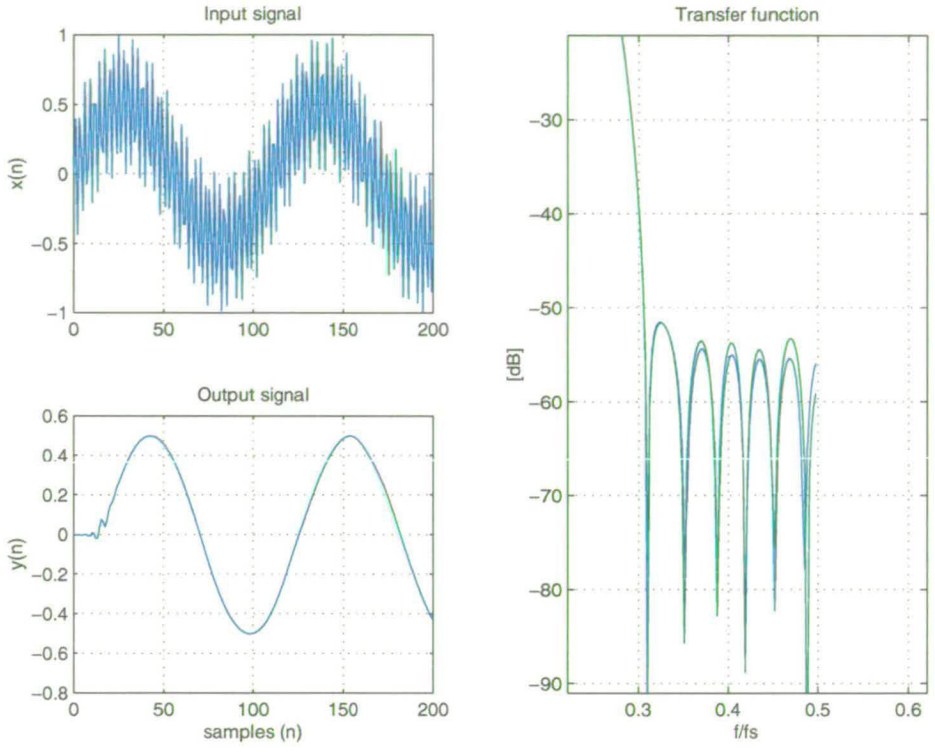


Figure 6.1: *Transfer function for 31-tap low-pass FIR Filter*

using the genetic algorithm described in section 5.2. Ten randomly generated populations of configurations-strings were created for each investigation; such that each PLA and FPGA topology evaluated is initially configured using the same set of configuration strings. This provides a common basis for comparison between all input, output and interconnect topologies, and between the FPGA and PLA architectures themselves. It is then the task of the genetic algorithm to manipulated each PLA/FPGA topology and generate the correct set of filter coefficients detailed in Table 6.1.

Communication between both the FPGA and PLA-based programmable platforms and the genetic algorithm is detailed in section 5.2 and illustrated in Figures 5.1 and 5.3. The entire EHW platform, comprising either the FPGA or PLA-based programmable PALU topology, the genetic algorithm and FIR filter coefficient parameters, is then simulated in detail using Cadence's *Leapfrog* VHDL simulation environment, where the best configuration string and corresponding coefficient fitness is written to file each generation.

A total of 6700 generations were performed by the GA for each of the 10 investigations, and for every programmable topology. The limit on the number of generations reflects the maximum

number of iterations each EHW platform can execute in one second of simulated "real time". One second was chosen as it was deemed the maximum period acceptable for adapting the filter specification, either due to component damage, or to modifications to the filter application.

6.3 Field Programmable Gate Array (FPGA) Topology

The FPGA developed in this thesis has been tailored specifically for implementing reduced complexity primitive operator filters, by replacing the FIR multiplication unit with a programmable series of bit-shifts, additions and subtractions. The PALU illustrated in Figure 5.2 therefore reflects the computational aspect of the CLB which is required to implement the coefficient multiplication stage of an FIR filter.

6.3.1 Interconnecting CLBS for an FPGA-based FIR Filter

There are a number of simplifications which can be made to the nearest neighbour connection topology highlighted in Figure 4.10(a). Because an FIR filter must be stable, no feedback between CLBs can be permitted as this might cause the filter configuration on the FPGA to become unstable. As a result each CLB will receive data from the south and east of the array, and output data from the north and east. Data travelling westward across the CLB array is therefore not permitted. This was also done to constrain the number of possible configurations on the FPGA, thereby reducing the search space required to find an acceptable filter solution, and lessening the burden on the genetic algorithm. Figure 6.2 illustrates the CLB element which incorporates the FPGA-based FIR filter.

Programmable routing is performed by six 2:1 Multiplexor units, each governed by a single control bit. $C10$ and $C9$ determine which of the two inputs $HrzIN$ (east input), or $VrtIN$ (south input) are passed to the PALU. $C8$ and $C7$ then controls whether the output of the PALU is fed into the final routing unit, or whether the CLBs original inputs are to be selected. If this is done then the CLB performs a through connect operation. The output routing of each CLB is determined by control bits $C1$ and $C0$ (bits $C6 - C2$ are used to configure the PALU). $HrzOUT$ and $VrtOut$ form the output of each CLB, which then connect to $HrzIN$ and $VrtIN$ of the next CLB, determined by one of the the interconnect topologies detailed in Figure 6.3.

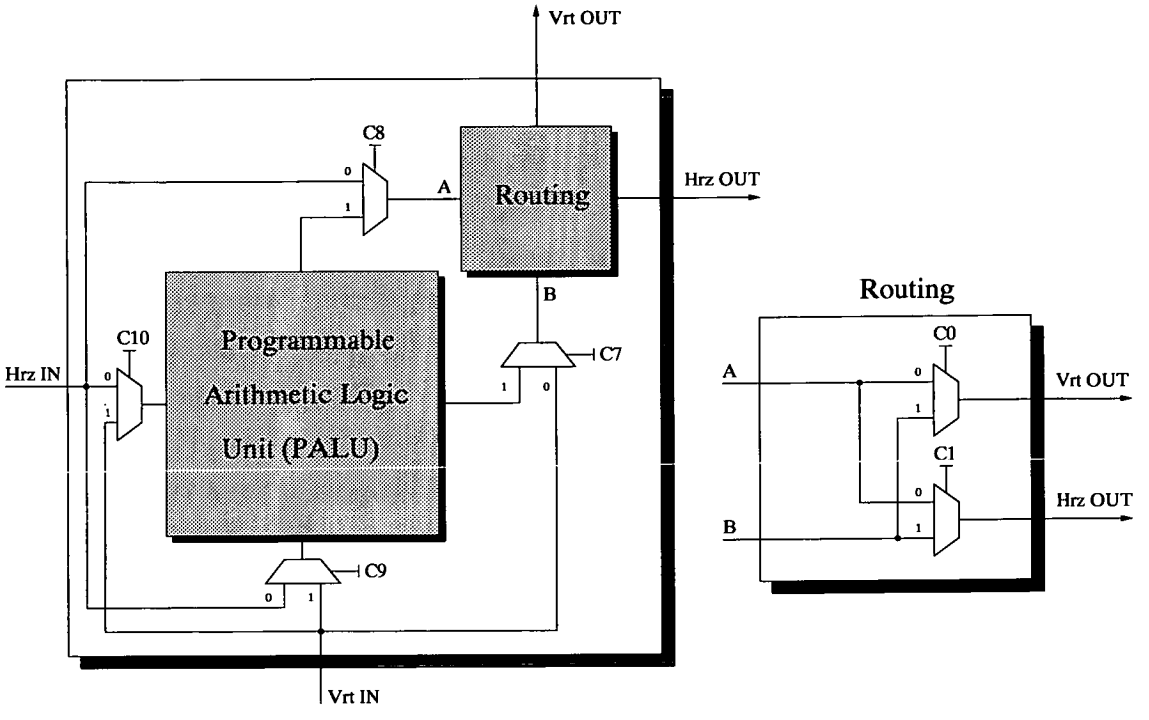


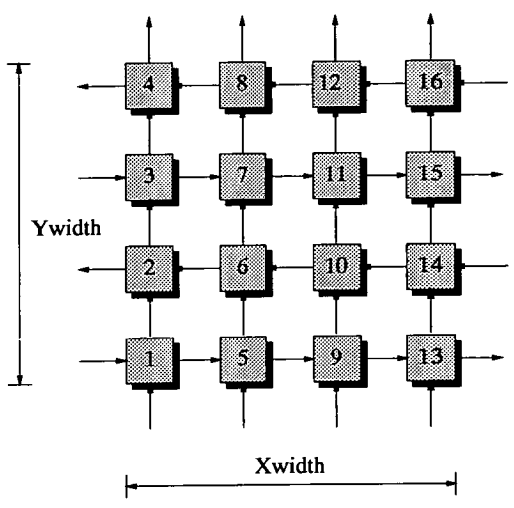
Figure 6.2: *Configurable logic block (CLB) for FPGA including routing to and from PALU.*

The three interconnect topologies, shown in Figure 6.3, were investigated based on nearest neighbour connectivity to determine the most suitable interconnect sequence for implementing FIR coefficient multiplication on an FPGA-based EHW platform.

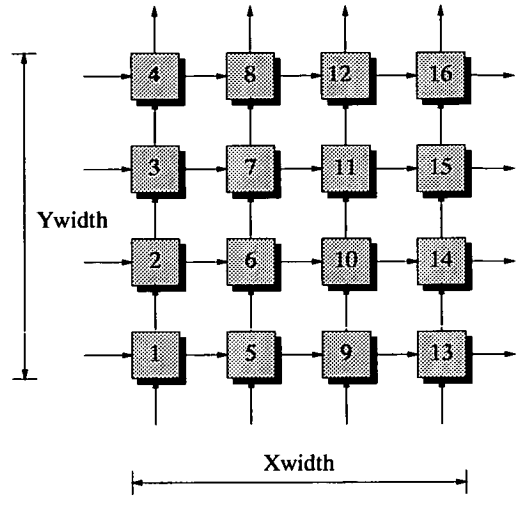
- AFFA: Alternating feed-forward array. The flow of horizontal inputs fed to each CLB alternates from east to west. Although westward data flow is permitted in this topology it is still constrained such that no CLB feedback is possible. This interconnect topology was designed to maximise linkage between PALUs by providing a maximally long critical path through the PALU array.

- CFFA: Continuous feed-forward array. Similar to a systolic array such that each PALU is clocked and data flows from the bottom left CLB of the FPGA to the top right.

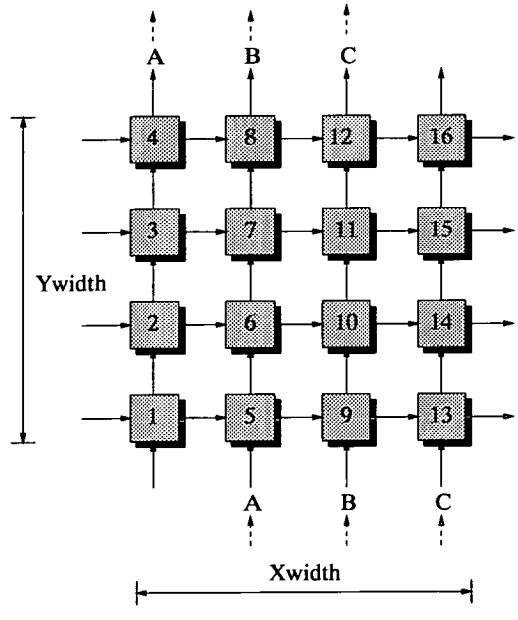
- CFFLA: Continuous feed-forward loop array. The connection topology builds on the CFFA by permitting connectivity between CLBs on the top row of the FPGA, with the CLB of the next adjacent column on the bottom of the FPGA. Again this approach eliminates any contentious configurations resulting from feedback whilst providing high connectivity.



(a) Alternating feed-forward array topology.



(b) Continuous feed-forward array topology.



(c) Continuous feed-forward looping array topology.

Figure 6.3: *Various routing topologies for interconnecting PALUs in FPGA structure.*

It is also important to determine the optimal placement of filter taps within the FPGA architecture such that each FIR filter can be implemented successfully in hardware and with minimal CLB resource. Four programmable output topologies for placing FIR coefficient taps have therefore been investigated.

- EOS: Edged output sequence. All CLBs on the outer edge of the FPGA are potential filter taps as shown in Figure 6.4(a). Each filter coefficient can therefore be generated by programming the relevant output CLB. The disadvantage of this approach is that a number of CLBs within the array must act as through connects to those CLBs on the outside edge. The total number of CLBs available as potential tap outputs is therefore

$$EOS_{taps} = (Ywidth * 2) + ((Xwidth2) - 4) \tag{6.1}$$

  Where $Xwidth$ and $Ywidth$ represents the number of CLB columns and rows respectively.

- AOOS: Alternating orthogonal output sequence. The topology shown in Figure 6.4(b) enables filter taps to be output throughout the CLB array. This approach was intended to reduce the need for CLB though connect which might arise in the EOS topology. The total number of CLBs available as potential tap outputs is given by

$$AOOS_{taps} = (Ywidth/2) * Xwidth \tag{6.2}$$

- AAOS: Alternating arrow output sequence shown in Figure 6.4(c) is a derivative of the AOOS topology. However AAOS provides better localised connectivity between potential output CLBs, which more tightly couples the generation of partial products required to produce subsequent tap outputs within the coefficient set. The total number of CLBs available as potential tap outputs can be calculated as

$$AAOS_{taps} = (Ywidth + 1) * (Xwidth/2) \tag{6.3}$$

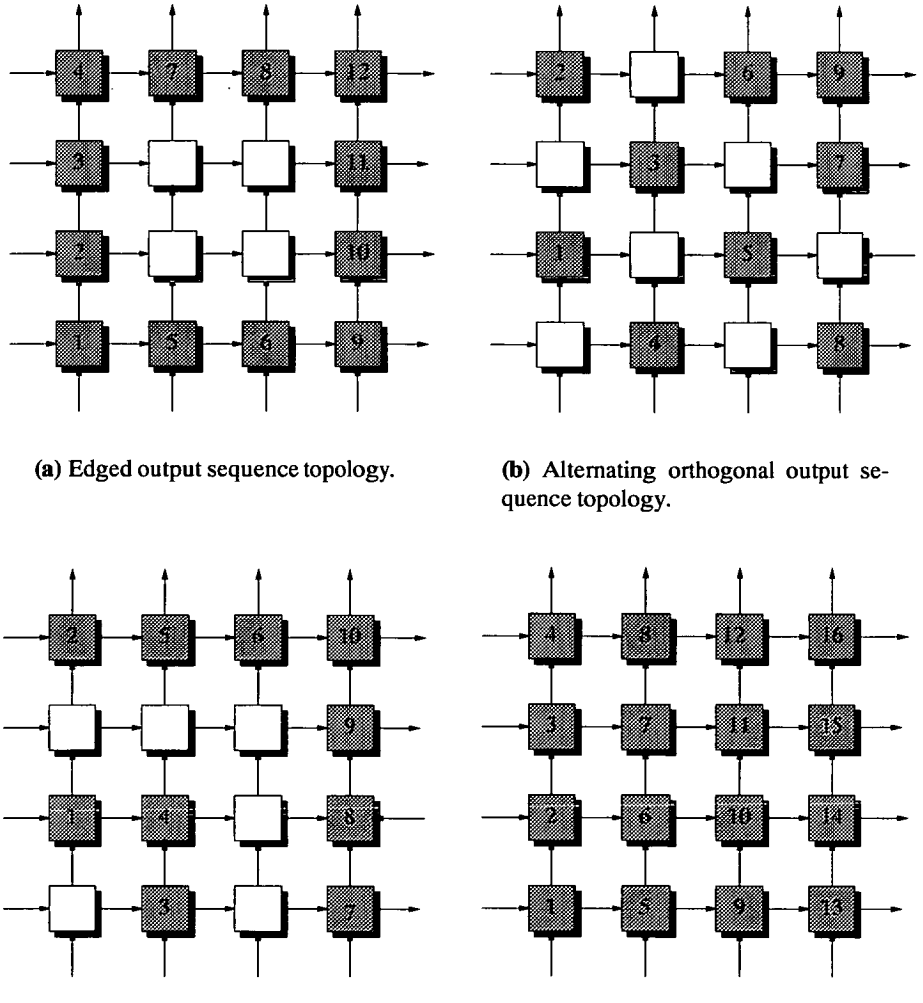
  Both AAOS and AOOS topologies provide the almost the same number of output CLBs.

- BLOS: Base-line output sequence. Whilst the topology shown in Figure 6.4(d) is highly unrealistic in terms the high degree of control logic and interconnect that would be required to implement in hardware, it provides the genetic algorithm with a highly flexible

means of implementing the desired filter response as every CLB in the array is a potential filter tap. The number of available CLBs is therefore

$$BLOS_{taps} = Ywidth * Xwidth \tag{6.4}$$

The number of bits required to encode the allocation of a CLB to a given tap is determined by the number of CLBs which can potentially output a coefficient tap. For example, if a 4x4 array of CLBs utilised the base-line output sequence, BLOS, then 4-bits would be required to encode the relevant tap on each output CLB in the range 0 to 15.



(a) Edged output sequence topology.

(b) Alternating orthogonal output sequence topology.

(c) Alternating arrow output sequence topology.

(d) Base-line: All CLBs connected output sequence topology.

Figure 6.4: *Various output topologies for FPGA structure.*

In addition to CLB interconnect and output topologies, the connectivity and control of the filter input, $X(n)$ must also be considered. Two signal input topologies are therefore presented. Figure 6.5 illustrates the L-shaped input sequence (LSIS) as it would be in conjunction with the CFFLA interconnect topology.

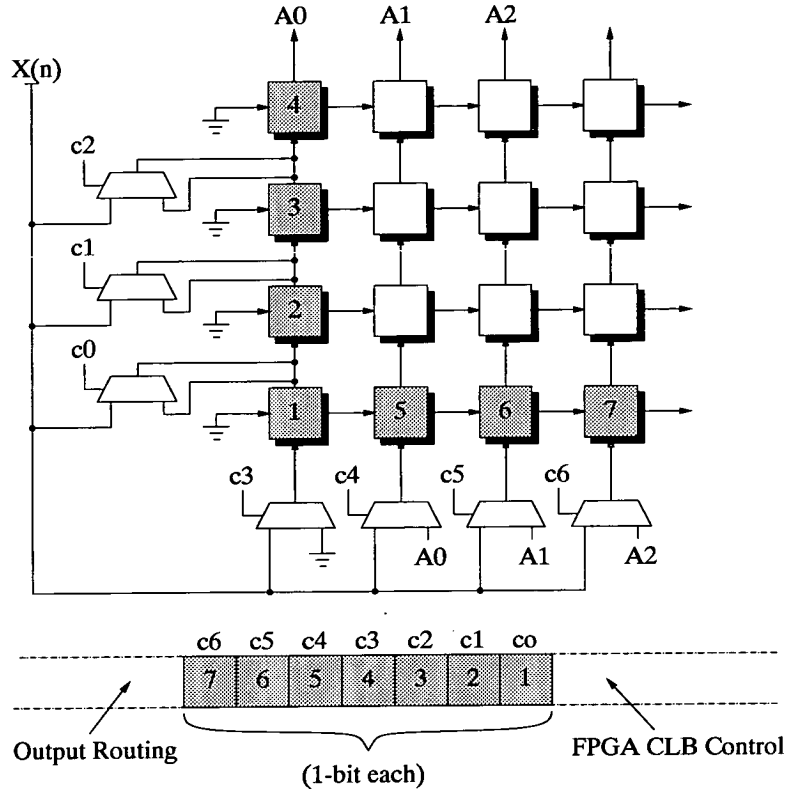


Figure 6.5: *FPGA control of FIR filter input $X(n)$; including position of input control string within FPGA string encoding.*

Because all of the FPGA-based interconnect topologies presented in this thesis are feed-forward, $X(n)$ is connected via Multiplexor control only to CLBs on the far right column of the array, and the bottom row. CLB inputs not directly connected to either a neighbouring CLB or $X(n)$ are pulled low (given a logic value of zero). The total of number control bits required to determine the input connectivity of $X(n)$ is then

$$I = log_2 Y width + (X width - 1) \tag{6.5}$$

The second input topology is considered the base-line input sequence (BLIS). This configuration connects all horizontal CLB inputs to $X(n)$ via Multiplexor control. All north feeding

CLB inputs therefore receive an input from either their nearest southerly neighbour, or from the filter input response. The total number of input control bits required is therefore

$$I = log_2 Y\,width * X\,width \tag{6.6}$$

6.3.2 Configuring the FPGA-based FIR Filter

Each FPGA-based filter architecture is configured via a binary configuration string. Each bit string is compartmentalised into three regions of control, defining the FPGAs output routing, input routing and individual CLB configuration as illustrated in Figure 6.6.

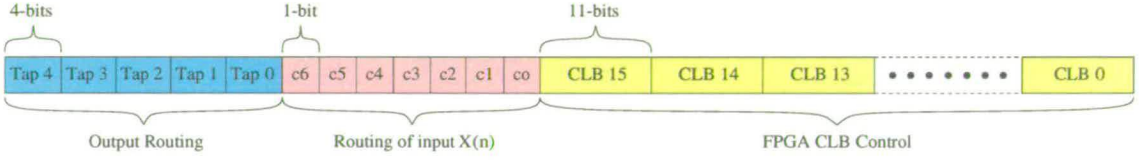


Figure 6.6: *Example configuration string for 4x4 FPGA-based FIR filter with LSIS, AFFA and EOS.*

The total length of configuration bit string can therefore be calculated as follows:

$$S_{FPGA} = log_2 O_c + I_c + (11 * X\,width * Y\,width) \tag{6.7}$$

Where O_c is the total number of output bits governed by the output topology expressed by the relevant equation from (6.1) to (6.4), I_c is the total number of control bits used to program the filter input $X(n)$ determined by the current input topology defined in equation (6.5) or (6.6), and S_{FPGA} is the total resulting bit length required to program the FPGA.

Figure 6.7 presents an example FPGA configuration of the 5-tap primitive operator filter originally illustrated in Figure 4.9. A 4x4 CLB array is interconnected using the AFFA topology, with filter taps connected to CLBs using the EOS. The input pulse is held constant at logic '1' and connected to the FPGA via the L-shaped input sequence (LSIS) in order to produce the desired coefficient set. The bit string required to configure the FPGA-based FIR filter is also shown, and has been sectioned into the three regions of control discussed above. A number of the control bits used to configure the CLBs shown in Figure 6.7 are in the "don't care" state, 'x'. This due to redundancies inherent in the CLB control encoding. For example when a CLB acts as a simple though connect, as is the case with the CLB at position 0 in the array, then the

control bits used to encode the PALU become redundant, as do control bits $C10$ and $C9$ which govern the inputs to the PALU.

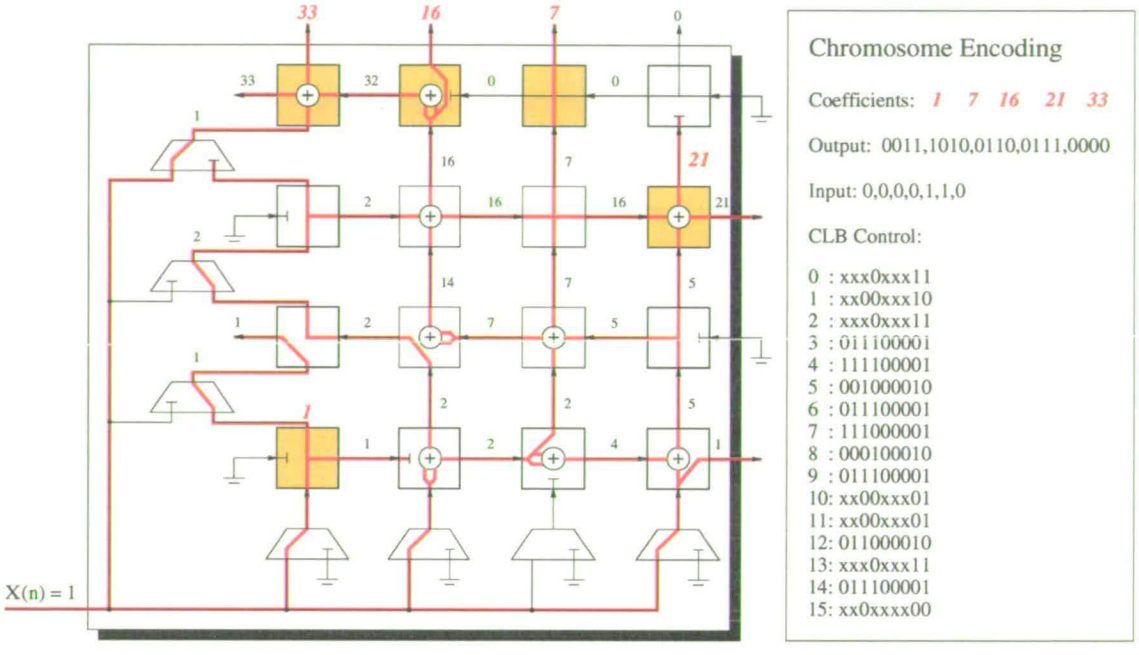


Figure 6.7: *Example FPGA configuration of 5-tap primitive operator filter.*

6.3.3 FPGA-based FIR filter Parameters

An 8X8 array of PALUs was chosen to implement the 31-tap filter presented in section 6.2. 64 PALU elements were deemed to be sufficient to provide enough partial product terms to generate the 9 distinct coefficient taps that were required. The dimensions of the FPGA were kept symmetrical so that each of the tap output topologies could optimally utilise the four interconnect sequences. The word length of the filter input, $X(n)$, the FPGA coefficient output taps, and the PALU processing elements were parameterised to 16-bits and represented in 2's compliment within the VHDL model. These parameters were set to match the specification of the low-pass FIR filter. The FPGA filter input, $X(n)$ is held constant at a value of one (i.e. "0000000000000001" in 16-bits) so that each selected tap output can be compared directly with the corresponding coefficient in the filter specification. A clock of 50MHz was used to control the FPGA-based EHW platform so that approximately 6500 generations were performed within the one second evolution window specified. A total of 831 configurations bits are therefore required to implement the 8x8 FPGA-based filter when implemented with base-line input and

base-line output topologies (BLIS and BLOS respectively). This represents a search space of 2^{831} possible bit string configurations which must be successfully manipulated by the GA in order to achieve the desired low-pass filter response.

6.3.4 Investigation of Genetic Operator Parameters

Both the crossover and mutation rates of the genetic algorithm developed in section 5.3 are parameterisable, and are user defined before circuit evolution. A crossover rate of 0.6 (60%) is generally recommended [53], and is consistent with the rate set for the Virtual Chip EHW platform in Chapter 3. However, the parameterisation of genetic operators depends greatly on the problem domain, and the manner in which the chromosome is encoded. These two constraints differ significantly from the GA implementation required for the Virtual Chip. As a result two crossover rates were initially examined. The first utilised a crossover probability of 0.6, the second removed crossover altogether (probability 0.0). This was done in order to determine the effectiveness of crossover as a search mechanism for determining an acceptable FPGA filter configuration using a binary encoded chromosome.

All three FPGA interconnect topologies were evaluated when evolving the low-pass filter with and without the crossover operator. This was to achieve a thorough indication as to the usefulness of the crossover operation. In order to maintain continuity over each evaluation, the L-Shaped Input Sequence (LSIS), and Edged Output Sequence (EOS) were arbitrarily selected as the I/O topologies. The probability of mutation was set according Mühlenbein's 1 over bit length rule, defined in equation (2.1).

The fitness of each configuration string is calculated by the EHW platform using equation (5.1). Each fitness score therefore lies in the range 0.0 to 9.0. This corresponds to how effectively each FPGA topology was mapped to produce the 9 distinct filter coefficients required to implement the desired low-pass transfer function shown in Figure 6.1; and represents a functional correctness of 0 and 100% respectively. All other run-time parameters are specified above. Results showing the average fitness of each low-pass filter evolved with and without crossover over 10 evolutionary runs are displayed in Table 6.2.

The average fitness of each filter coefficient set evolved on the three FPGA interconnect topologies does not seem to be influenced by the presence or absence of crossover. In fact, the fitness of the coefficients evolved using the GA without crossover is marginally higher than the

Connection Topology	Average Fitness with Crossover	Average Fitness without Crossover
AFFA	8.54595 = 95.1%	8.6358 = 96.0%
CFFA	8.59490 = 95.5%	8.6124 = 95.7%
CFFLA	8.64150 = 96.0%	8.6501 = 96.1%

Table 6.2: *Performance of FPGA connection topologies in generating the 31-tap low-pass filter configured using genetic algorithm with and without crossover.*

corresponding coefficient set generated when crossover was employed. This is almost certainly due to the high level of epistasis inherent in the POF design problem, such that interactions between PALU elements and interconnects (expressed as genes in the chromosome) are non-linear and filter fitness cannot be directly attributed to the effects of an individual gene. Maximum epistasis then occurs when the fitness contribution of an individual gene depends on the values of all other genes in the chromosome, resulting in a highly uncorrelated search space. High degrees of epistasis therefore inhibit the effectiveness of crossover as a search mechanism through chromosome recombination of two parent solutions. As the problem presented to the GA becomes increasingly difficult (epistatic) crossover become less effective and the search favours progress though mutation [82, 127, 128]. Crossover is therefore no longer applied to filter coefficients evolved using the FPGA-based EHW platform.

In order to ascertain the influence of varying the mutation rate, P_m, equation (2.1) was modified to produces two new probabilities of bit-flip mutation 2 and 3 times greater than that originally used: $P_m = 2/N$ and $P_m = 3/N$ respectively. The same evolutionary runs as above were performed, the results of which are presented in Table 6.3. Crossover was not employed.

Connection Topology	Average Fitness at $P_m = 2/N$	Average Fitness at $P_m = 3/N$
AFFA	8.6538 = 96.0%	8.4132 = 93.4%
CFFA	8.5949 = 95.5%	8.2569 = 91.7%
CFFLA	8.7543 = 97.3%	8.6880 = 96.5%

Table 6.3: *Performance of FPGA connection topologies in generating the 31-tap low-pass filter configured using genetic algorithm with variable mutation rates.*

The results show very little difference between the average fitness of the filter coefficient sets generated by the GA using the increased mutation probabilities. Only the CFFLA interconnect topology produced a noticeably better solution (97.3%) when $P_m = 2/N$. As a result the

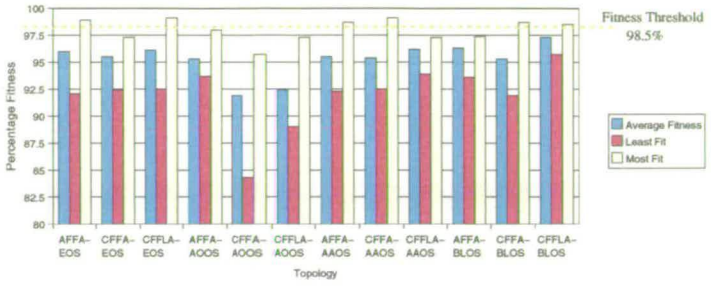
original mutation rate $P_m = 1/N$ will be maintained, as this will conform with excepted GA practices.

6.3.5 Performance Comparison of FPGA Topologies

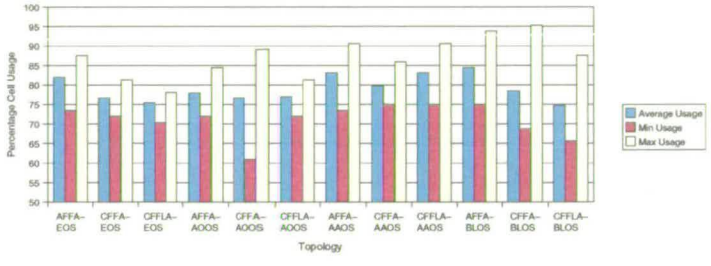
The performance of each FPGA input, output and interconnect topology was assessed based on four criteria: The fitness of the filter coefficient set, the number of PALUs used, the degree of PALU re-use (generation of partial product terms), and the total number of shift, add and subtract operations required to implement the specified low-pass filter coefficient set. All three interconnect topologies and all four FPGA tap output sequences were initially investigated using the LSIS. The results of each performance criteria, averaged over ten evolutionary investigations are shown in Figure 6.8.

Coefficient fitness: Analysis of the transfer functions produced by the evolved FPGA coefficients sets reveals that a fitness score of $\geq 98.5\%$ is required to produce an acceptable low-pass characteristic, with a gain no less than -52 dB. An example of the minimum filter performance criteria produced at 98.5% is provided by the green transfer function illustrated in Figure 6.1. This highlights the small margin for error which can be accepted in order to successfully generate the highly constrained filter specification outlined in Table 6.1. Fitness variations of between 0.5 and 1% are therefore critical and highly challenging for EHW. The GA was able to produce an acceptable coefficient set on each output topology except AOOS. In fact the AOOS output topology produced the worst set of coefficients regardless of the interconnect topology. The CFFLA interconnect topology provided the GA with the greatest PALU connectivity and as a result produced on average the fittest coefficient sets across each of the output topologies (with the exception of AOOS) It is therefore not surprising that CFFA interconnect, which exhibits the least PALU connectivity as defined by shortest critical path through the FPGA, produced on average the poorest results.

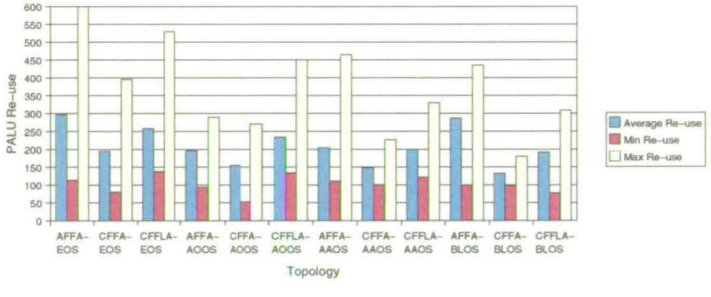
On average, the best coefficients sets produced by the GA were attained using a CFFLA-BLOS interconnect and tap output combination. However, the fittest coefficient set was generated using the CFFLA-OROS FPGA topology with a 99.1% coefficient match. No FPGA topology was able to provide the GA with a means of producing a coefficient set that matched exactly with the original. However, the very nature of FIR filter design allows for some compromise, for example due to implementation factors such as coefficient quantisation error. The EHW approach to FIR filter implementation therefore introduces some error, however, it can be seen



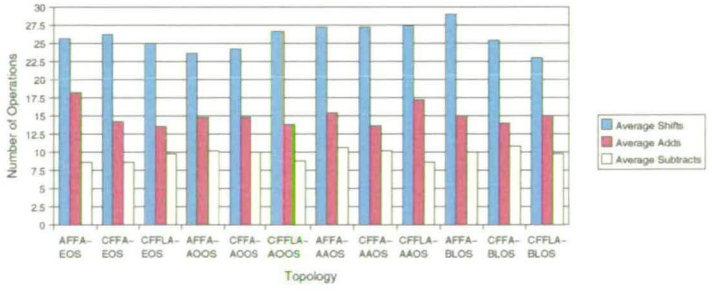
(a) Success of FIR filter based on fitness criteria.



(b) PALU cell usage (coverage) required by FIR filter.

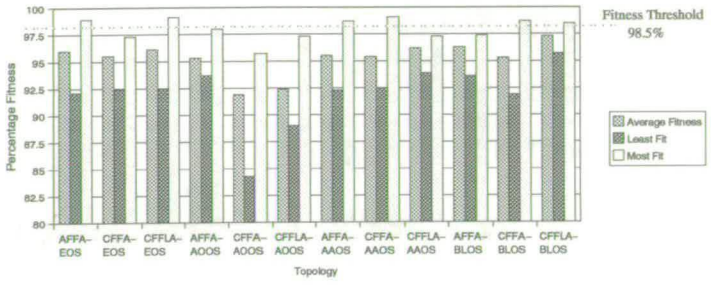


(c) PALU re-use exploited by FIR filter.

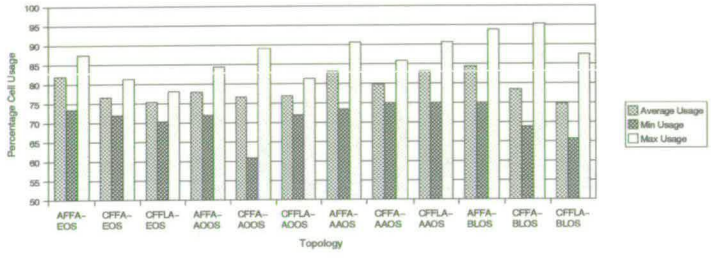


(d) Operations performed by PALU to implement FIR filter.

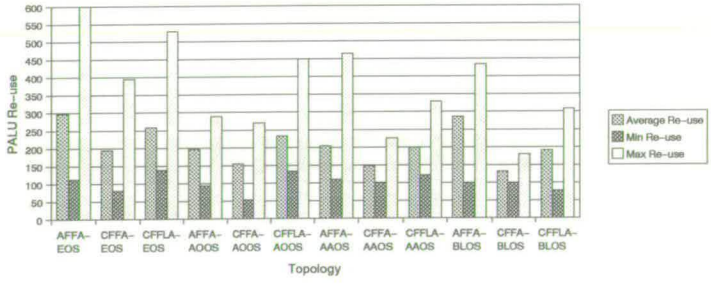
Figure 6.8: *Performance of various FPGA interconnect and coefficient output topologies to autonomously generate a 31-tap low-pass FIR filter. L-shaped input sequence (LSIS) employed.*



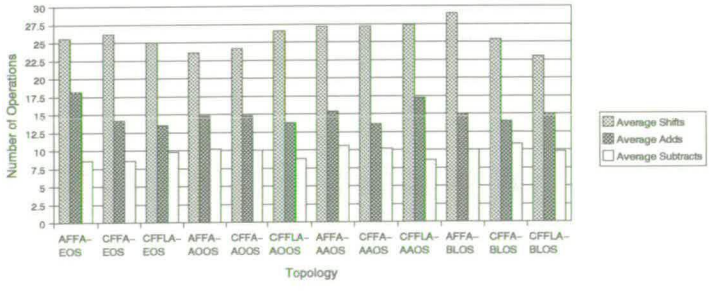
(a) Success of FIR filter based on fitness criteria.



(b) PALU cell usage (coverage) required by FIR filter.



(c) PALU re-use exploited by FIR filter.



(d) Operations performed by PALU to implement FIR filter.

Figure 6.8: *Performance of various FPGA interconnect and coefficient output topologies to autonomously generate a 31-tap low-pass FIR filter. L-shaped input sequence (LSIS) employed.*

that these are often minimal, and still provides acceptable solutions.

PALU utilisation: The average number of PALUs utilised within the FPGA remains relatively constant at about 78% of the total FPGA area, regardless of interconnect or output topology. PALU coverage therefore seems to have little influence over coefficient fitness. However, a link can be established with the AOOS output topology, which implements the fewest PALUs when averaged across all three interconnect topologies, and also exhibits the poorest filter coefficients.

The AFFA interconnect topology promotes the greatest PALU utilisation regardless of the output topology. This can be explained by the manner in which PALUs are connected within the AFFA. Both the AFFA and CFFLA interconnect topologies produce the same critical delay path in the FPGA which lies between the bottom left PALU and top right. This delay is then equal to the number of PALUs in current the architecture (for the 8x8 array employed, this is 64). However, many more considerably shorter paths can be achieved between these two points using the CFFLA. This is because all PALUs are routed in the same direction. However, the AFFA is almost forced to implement more PALUs by virtue of its routing topology and close PALU linkage. The AFFAs more constrained routing might be reduced by providing interconnectivity between PALUs on the top and bottom rows, as with the CFFLA. However, this would result in an architecture which exhibited feedback, a property which is not desired in linear FIR filtering.

PALU re-use: Again there appears to be little correlation between the degree of PALU re-use and coefficient fitness on any of the FPGA topologies. However, a relationship can be established between PALU re-use and utilisation on FPGAs which utilise AFFA interconnect. Both PALU re-use and utilisation are at there highest when the AFFA is employed, and is the case for each output topology examined. This is again due to the high degree of linkage between PALUs connected using the topology. The lowest degree of PALU-reuse is exhibited by the CFFA interconnect sequence, regardless of the FPGA tap output topology. This is because, of all three FPGA interconnect topologies examined, the CFFA provides the lowest linkage between neighbouring PALUs, and also exhibits the shortest critical path ($(Xwidth - 1) + Ywidth$ PALUs).

PALU operations: In all of the FPGA topologies examined, the number of shift operations selected by the GA is approximately 40% higher than the number additions. This is partic-

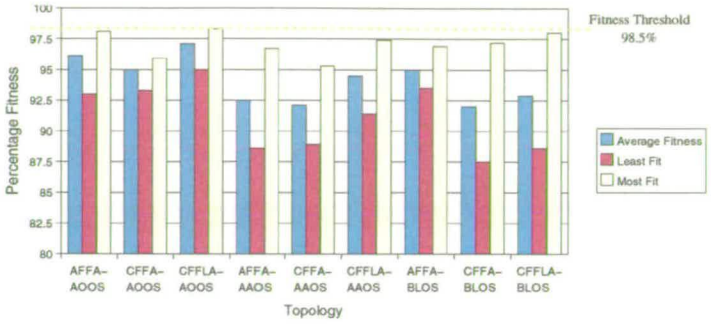
ularly desirable as shifts consume almost no power and have low area, and are the primary means of partial product generation using multiplierless design techniques such as CSD and POF. Approximately twice as many PALUs implement addition than subtraction. Again this is encouraging for an EHW platform based around the POF design because Bull has shown subtraction to play a relatively minor role [2], which is reflected by the genetic algorithms configuration of PALUs within the FPGA-based filter.

Figure 6.9 presents the following set of results obtained after the GA was again used to examine three interconnect topologies and both FPGA output sequences, this time with the BLIS input topology in place of the LSIS.

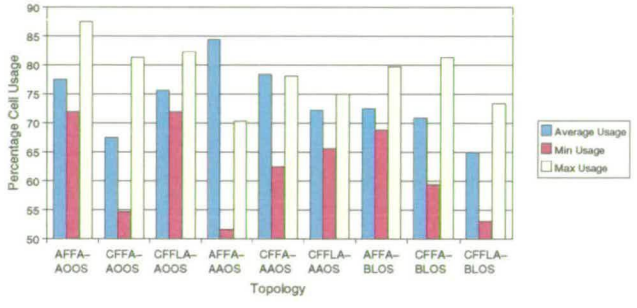
Coefficient fitness: The inclusion of the Base-Line Input Sequence (BLIS) generates a number of difference in the quality of filter coefficients produced by the GA on each of the FPGA topologies, where the L-Shaped Input Sequence is substituted. The most noticeable difference is the success of the AOOS output topology using BLIS when compared to LSIS. The AOOS now provides the GA with the only topology capable of producing an acceptable filter coefficient set ($\geq 98.5\%$) within the specified evolution time. This was achieved using CFFLA interconnect. It can also be seen that all FPGA topologies which utilise AOOS produce coefficients with significantly better fitness (as much as 4% between CFFLA-AOOS and CFFLA-BLOS) than that attainable using other output sequences. Regardless of input or output topology the CFFA again produces the poorest coefficient sets.

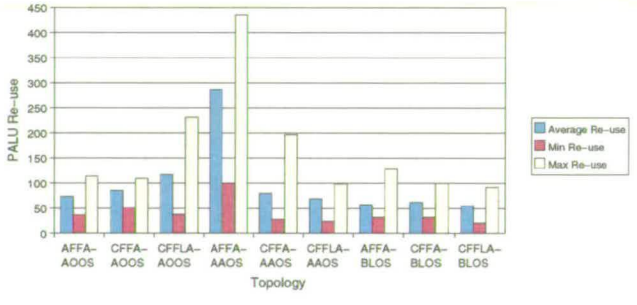
The generally poor performance of FPGA topologies configured using BLIS can be attributed to the increased complexity of the configuration string and a resulting increase in the epsistatic representation of the chromosome. A further 59 bits are required to encode BLIS compared to LSIS, each relating to any given PALU in the array (see equations 5.9 and 5.10). This further complicates the search space and can disrupt routing between horizontally connected PALUs, due to the manner in which the BLIS is connected.

PALU utilisation: As with the LSIS input topology, the AFFA interconnect sequence continues to promote the highest PALU utilisation when configured by the GA. On average fewer PALUs are used to implement coefficient sets when interconnects are configured with base-line input and output topologies. This is most likely due to the freedom of input and tap positioning afforded by these approaches, thereby lessening the need for extensive PALU connectivity and re-use required by the other input/output topologies.

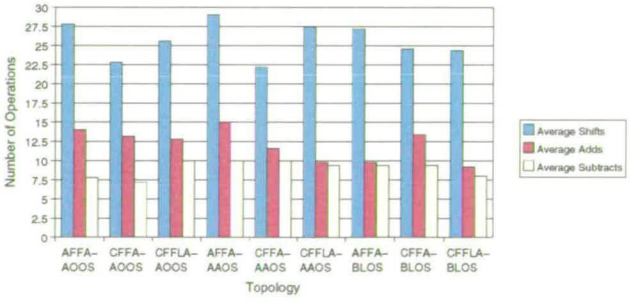


(a) Success of FIR filter based on fitness criteria.





(c) PALU re-use exploited by FIR filter.



(d) Operations performed by PALU to implement FIR filter.

Figure 6.9: *Performance of various FPGA interconnect and coefficient output topologies to autonomously generate a 31-tap low-pass FIR filter. Base-line input sequence (BLIS) employed.*

PALU re-use: On average FPGA topologies utilising BLIS re-use 50% fewer PALUs than equivalent FPGA topologies implementing LSIS. This is again due to a reduction in dependency (linkage) between neighbouring PALUs, resulting from the universal availability of the filter input, $X(n)$. It is unclear as to why the AFFA-AAOS FPGA topology promotes considerably higher PALU re-use than the other output and interconnect topologies using BLIS (in fact re-use is also 50% greater than with the equivalent FPGA topology using LSIS). However, the fact that AFFA interconnect again promotes such high levels of PALU re-use further supports the observation that extensive PALU linkage (inherent in AFFA interconnect) translates to high PALU re-use.

PALU operations: Similar trends between shifting, addition and subtraction can be seen between output and interconnect topologies using BLIS as with those using LSIS.

6.3.6 Graphical Representation of FPGA-Based FIR Filter

In order to represent the results obtained in section 5.5.5 above, graphical software was developed to visualise the FPGA topologies implemented, and present the FPGA configurations programmed by the genetic algorithm. To achieve this three postscript templates were generated reflecting each of the three interconnect topologies investigated. Horizontal connections are coloured red, vertical connections green, and those connected to $X(n)$ are coloured blue. Each of these postscript templates can be found in Appendix B.

The functionality of each input, output and interconnect topology is coded in C, and the evolved configuration bit string (originally created and then saved by the VHDL model of the FPGA-based EHW platform) used as input to the C program. The selected configuration string is then used to generate a graphical postscript representation of the FPGA, displaying PALU operations and programmable interconnect. The C program generates postscript by appending additional information about the FPGAs configuration to the relevant postscript interconnect template. Each PALU operation is then scripted as a yellow coloured box containing the relevant symbol: +, -, or S1 to S4, relating to shift-left 1 bit to shift-left 4 bits respectively. Each PALU also displays four coloured regions relating to its two inputs and two outputs (i.e. top region = horizontal output, bottom region = horizontal input). A PALU which displays a yellow output region is therefore outputting the result of its arithmetic operation. PALUs with output regions the same colour as either the horizontal or vertical interconnect are denoted as acting through connects. The horizontal and vertical numerical output of each PALU is also shown, and coef-

ficient tap outputs are marked according to the taps position in the coefficients set. In addition, the corresponding output PALUs are highlighted. PALUs which serve no purpose are marked out in grey. Figure 6.10 displays the graphical representation of the most successful low-pass filter coefficient set evolved by the GA which was configured on the LSIS-CFFLA-EOS FPGA topology.

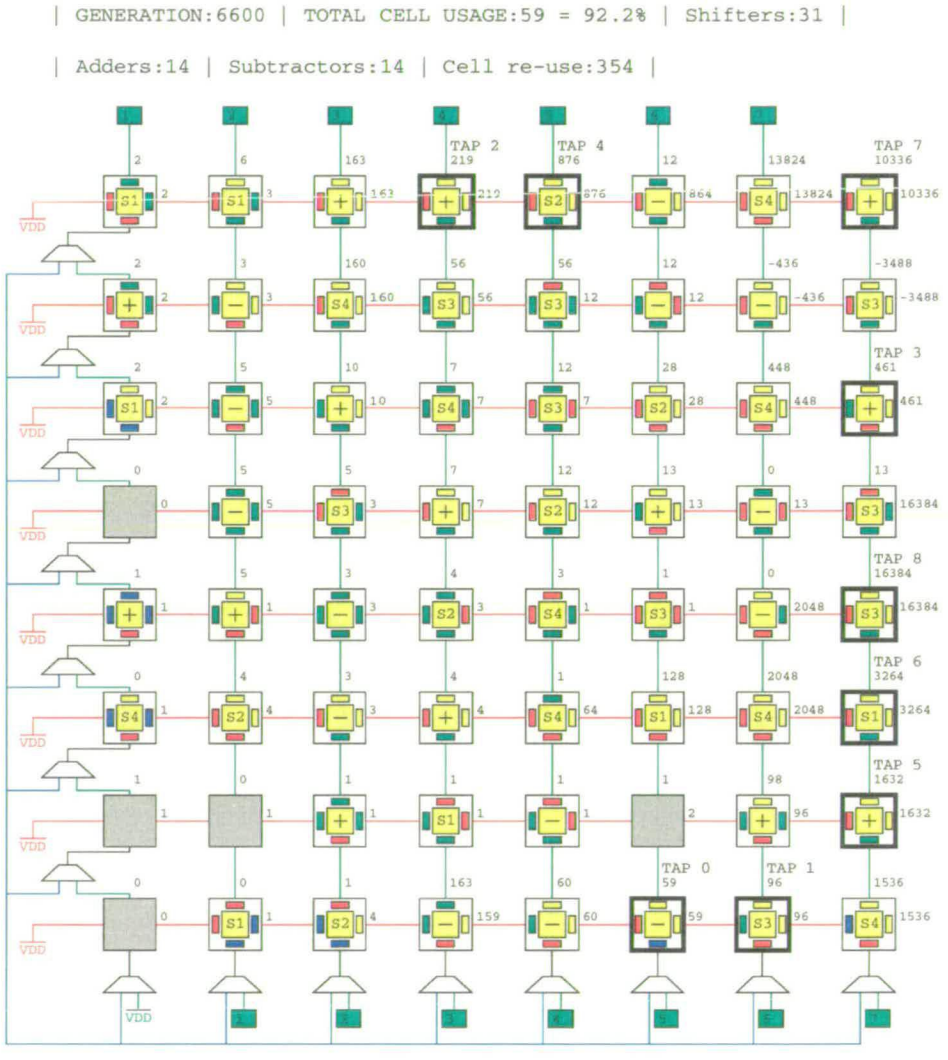


Figure 6.10: *Example FPGA configuration of 31-tap low-pass filter.*

The graphical software described was further developed to gain additional information about the contribution of each PALU in generating the desired coefficient set. The degree of PALU re-use was therefore mapped onto the relevant FPGA interconnect template and represented as a normalised RGB (Red, Green, Blue) colour scale, each in the range 0 to 255, where 255 represents the maximum of a given colour. A recursive tree search was used to trace the connectivity

of each PALU acting as a tap output and noting the PALUs used as the trace progresses. It is therefore possible for multiple tree searches to follow the same path, thereby using the same PALUs. As a result the most extensively re-used PALU for any given FPGA configuration is denoted in RGB as maximum red (255, 0, 0) and the lowest re-use in maximum blue (0, 0, 255). PALUs that were not used are again marked out in grey. Figure 6.11 displays the corresponding PALU re-use map of the LSIS-CFFLA-ORS FPGA topology shown in Figure 6.10.

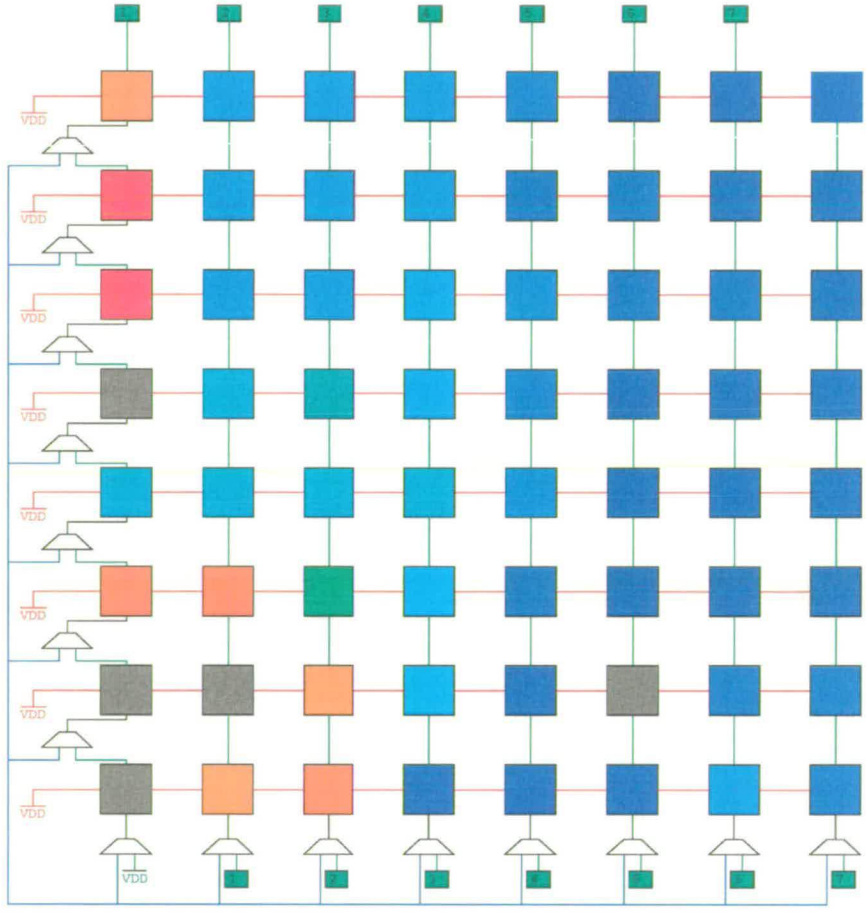


Figure 6.11: *PALU re-use map from FPGA configuration of 31-tap low-pass filter.*

It is interesting to note that the majority of PALU activity lies along the left side of the array, yet the majority of filter coefficients are output on the right of the array. This demonstrates that the partial product terms, necessary for coefficient generation, most frequently stem from terms generated by PALUs in the first few columns of the CFFLA interconnect topology. This matches well with the systolic nature of CFFLA, and the fact the filter input is only available to PALUs along the far right column and bottom row of the array.

obsolete. As a result the PLA configuration bit string requires no compartmentalization other than the identification of the left-shift-only PALUs present in the first column of the array. Each PALU and associated interconnect is therefore described as a binary word, W, in the order of PALU referencing shown in Figure 6.12; where PALU 1 denotes the LSB of the binary configuration string. The bit length of a binary word encoding a given PALU is thus described as

$$W = PALU_{cntrl} + 2R_c \tag{6.13}$$

Where $PALU_{cntrl}$ is the number of control bits required to program each PALU (5), and R_c is the bit length of the control required to encode each $P:1$ multiplexer in the programmable interconnect array. The total length of each binary configuration string required to encode a given PLA topology can then be described as

$$S_{PLA} = (Xwidth * Ywidth * W) + (Ywidth * S_c) + PLA_{taps} \tag{6.14}$$

Where $Xwidth$ is the number of PALU columns, $Ywidth$ is the number of PALU rows, W is the bit length of control required for each PALU and its interconnect, S_c is the number of bits used to determine the left-shift of PALUs in the first column, and PLA_{taps} is dependent on the output topology employed. Figure 6.14 describes the layout of the configuration string required to program the PLA. The region encoding the control of PALUs in the first column is shown in grey.

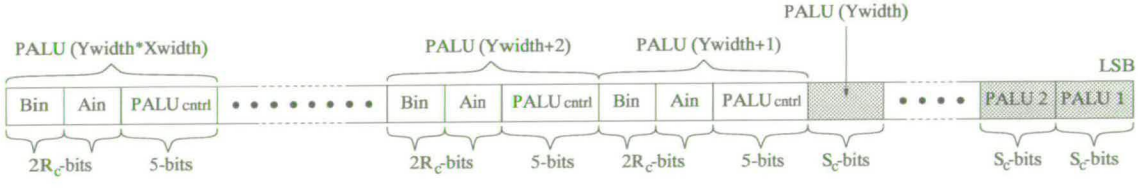


Figure 6.14: *Layout of configuration string for programming PLA.*

Figure 6.15 illustrates an example PLA configuration for implementing the primitive operator filter shown in Figure 4.9 using *Route 1* interconnect and *Output 2* tap routing. PALU's and interconnects which are not utilised are shaded in blue. As with the FPGA architecture, the input pulse, $X(n)$, is held constant at logic '1'.

It can already be seen that a great deal of redundancy is inherent in the PLA architecture, such

6.4 Programmable Logic Array (PLA) Topology

Like the FPGA-based architecture presented in section 6.3, the PLA described in this thesis has been developed specifically to implement reduced complexity multiplier-free coefficient multiplication for digital FIR filtering. The PALU element illustrated in Figure 5.2 again provides the backbone for the signal processing of $X(n)$ using the POF approach. Importantly, the PLA architecture is particularly suited to implementing the sum of products equation of (4.1) (required for FIR filtering) because of its inherent 2D array structure [129].

6.4.1 Interconnecting PALUs for an PLA-based FIR Filter

PALU's are arranged in columns, each column connecting to an array of interconnect logic, which in turn connects to the next column of PALU's. Every PALU in one column is thereby connected to every PALU in the next column via the interconnect array. Whilst costly in terms of physical area, this approach was initially taken to determine the most suitable *ideal connection topology*. Figure 6.12 provides an overview of the basic PLA-based FIR filter topology to be implemented.

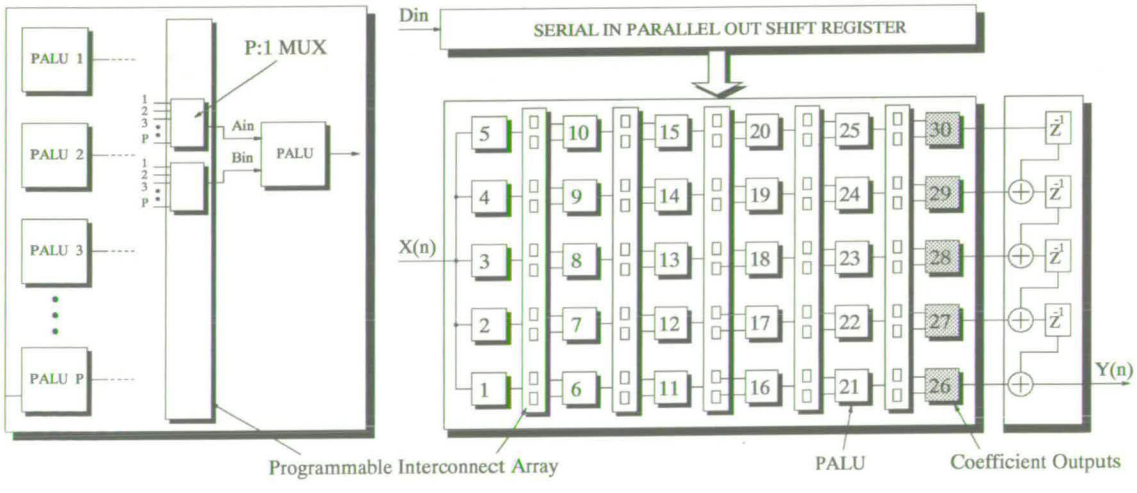


Figure 6.12: *PLA architecture and interconnect overview.*

Each interconnect array comprises a number of $P : 1$ Multiplexers which route the P outputs of the previous PALU column, where P denotes the number of PALUs per column. Both inputs of a given PALU are therefore connected to a separate $P : 1$ routing Multiplexer to provide maximum connectivity. A total of $2 * P$, $P : 1$ Multiplexers are required to construct each

programmable interconnect array. The number of PALUs in a column, $Y\,width$, and the number of columns in the array, $X\,width$, are determined in VHDL by the user during parameterisation of the PLA-based EHW platform. In addition to PALU routing, the interconnect array provides each PALU with a direct connection to the filter input, $X(n)$, and to logic '0'. This approach, originally implemented on FPGA topologies using BLIS, has been shown to reduce the number of PALU's utilised and re-used, thereby freeing up programmable interconnect resources. The number of control bits required to route each MUX is therefore given by

$$R_c = log_2(P + 2) \tag{6.8}$$

Where the addition of 2 to P represents the inclusion of routing for $X(n)$ and logic '0'.

PALUs are identified in numerical order, from the bottom left corner of the array to the top right corner. Due to the column based 2D topology of the general PLA architecture, all PALUs in the first, left-most, column are connect directly to $X(n)$. As a result, the first column of PALUs are left-shift only, as addition of $X(n)$ could only yield a result twice that of $X(n)$ (i.e. left-shift by 1), and subtraction would simply produce logic '0'. Therefore in order to provide additional functionality, all PALUs in the first column are capable of shifting between 0 and $L - 2$ bits, where L denotes the bit-width of $X(n)$. The number of control bits, S_c, required to determine the shift factor is then given by

$$S_c = log_2(L - 2) \tag{6.9}$$

It is important to determine the most suitable method of connecting PALU's to achieve high-performance FIR filtering using the PLA approach. As a result, four interconnect topologies were investigated and reflect various degrees of interconnectivity between columns of PALU as illustrated in Figure 6.13. These interconnect sequences vary from those investigated on the FPGA in that they go beyond nearest neighbour connectivity. Hierarchical interconnect has instead been investigated so as to maintain the column based form of the PLA architecture shown in Figures 4.11 and 6.12, whilst reducing the degree of interdependence (linkage) between PALUs, which has been shown to reduce the performance of filter coefficients generated on the FPGA. Additionally, routing sequences such as AFFA would simply turn the PLA into an FPGA-based topology. Each interconnect is categorised as follows:

- *Route 1*: Simplest interconnect sequence, and requires minimal routing. PALU's are only

connected to the next adjacent interconnect array (Figure ??). No feedback is therefore permissible. Routing to PALUs in non-adjacent columns is possible only by PALUs in intermediate columns performing a shift-by-zero. The number of control bits required to route each $P : 1$ MUX as part of the programmable interconnect array is given in equation (6.8).

- *Route 2*: 2-level interconnect; provides greater connectivity between PALU's in non-adjacent columns through additional routing between alternate interconnect arrays as shown in Figure ??. The number of control bits required to configure the routing multiplexers is therefore twice that of *Route 1* and is given by

$$R_c = log_2 2(P + 2) \tag{6.10}$$

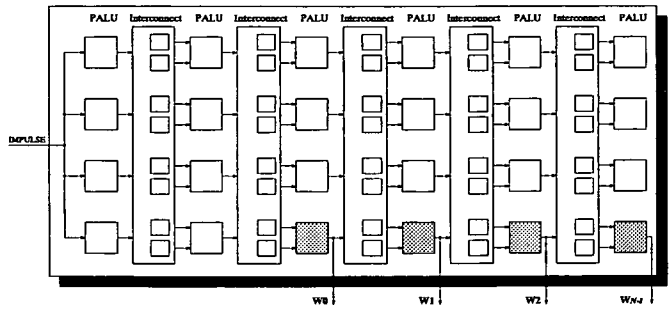
- *Route 3*: Utilises 4-level interconnect along with both 2-level and adjacent array connectivity, illustrated in Figure ??. This approach provides extensive connectivity between columns of PALU, intended to reduce linkage between adjacent PALU elements. However, the number of control bits required to configure each $P : 1$ MUX remains at that given in equation (6.10).

- *Route 4*: Comprises routes 1, 2 and 3 and additionally incorporates routing between neighbouring PALU's in the same column (Figure ??). This routing topology requires the most interconnect control for each routing Multiplexer. The number of control bits can be represented as

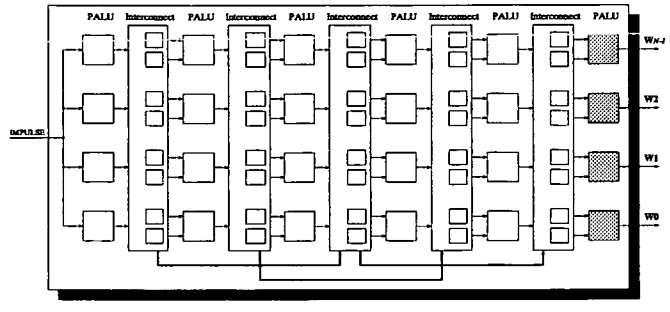
$$R_c = log_2 2(P + 3) \tag{6.11}$$

Optimal placement of filter taps within the PLA architecture is also important for generating an efficient FIR filter structure capable of meeting the demanding performance criteria required of many modern DSP applications. For this reason three output topologies for the placement of filter taps have been investigated and are identified as follows:

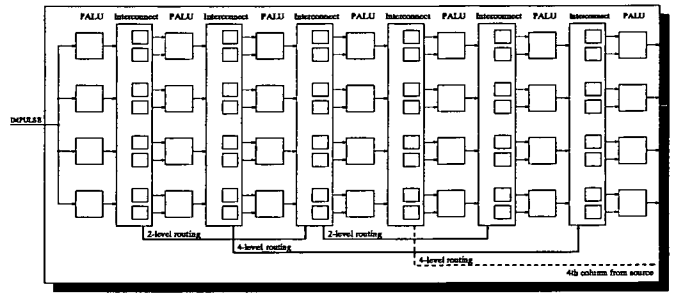
- *Output 1* employs row-based tap placement as shown in Figure ??. This topology is suitable for a direct form filter implementation in that coefficients can be summed and stored sequentially after each product term is generated in the correct order. To achieve this tap outputs must be ordered such that tap N (in the coefficient set 1 to N) is produced



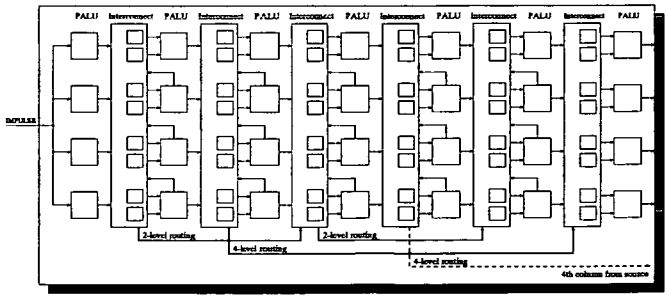
(a) Routing1: Localised connect.



(b) Routing2: Localised and 2-level connect.



(c) Routing3: Localised, 2 and 4-level connect.



(d) Routing4: Localised 2 and 4-level with column connect.

Figure 6.13: *Various Interconnect Topologies for PLA*

on the final, far-right, column in the PLA, and earlier coefficients in the set are present on columns progressively further from the final column. Note that taps are only connected to the bottom row. However, since all PALU's are ideally connected, the choice of row is irrelevant. A total of C clock cycles is required to process the filter input, $X(n)$, where C denotes the number of PALU columns and is the critical path. C therefore grows as the number of taps required increases. For any given filter specification only N output PALUs are therefore required.

- *Output 2* employs column-based tap placement as shown in Figure **??**. This topology is most suited to a transposed direct form filter implementation as all filter coefficients are present on the same clock edge. A total of C clock cycles is again required to process $X(n)$, however, C is no longer directly dependent on tap length, and could be considerably smaller than that required to implement *output 1* for complex filters with large coefficient sets. So as to utilise all PALUs, taps are output on the final column of the PLA such that coefficient 1 is output on the bottom row, with later coefficients output on PALUs at progressively higher positions in the column. As with *Output 1*, for any given filter specification of N taps, only N PALUs are required

- *Output 3* is classed as the base-line tap topology. Each PALU is capable of representing a filter coefficient. Whilst such a topology is highly unrealistic in terms of added control logic and interconnect, it provides the genetic algorithm with a highly flexible means of implementing the desired filter coefficient set and further reduces interdependency between PALUs. The total number of PALUs available as potential tap outputs is therefore given by

$$PLA_{taps} = Xwidth * Ywidth \tag{6.12}$$

Where a total of $log_2 PLA_{taps}$ bits are required to encode the desired filter tap at a given PALU location.

6.4.2 Configuring the PLA-based FIR Filter

Unlike the FPGA-based filter outputs presented earlier, tap outputs within the PLA are at fixed predetermined locations. This is because of both the nature of the PLA structure, and the hierarchical *ideal connection topologies* discussed above, which make programmable tap outputs

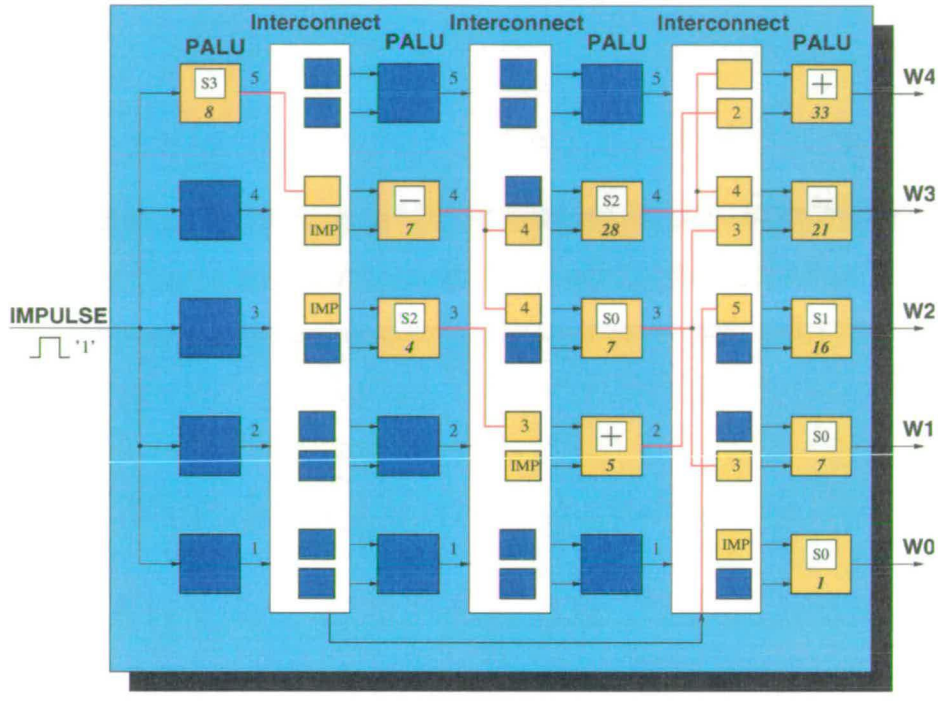


Figure 6.15: *Example PLA configuration of 5-tap primitive operator filter.*

that the topology is naturally suitable to fault tolerant design. This aspect will be examined more thoroughly in Chapter 7. Also note that a number of PALU's simply act as through connects (shift-by-zero) to adjoining PALU elements.

6.4.3 PLA-Based FIR Filter Parameters

Each PLA topology was investigated using 11 columns of PALU and 9 rows. These dimensions were chosen so that the 9 distinct taps of the low-pass filter could be configured using all three output topologies. Eleven columns of PALU where used (instead of 9) so that the first three columns could generate sufficient partial products to produce the desired filter coefficients when *output 2* was employed. It is therefore apparent that more PALUs are to be utilised using the PLA architecture than were implemented using the FPGA. However, larger PALU dimensions are required to ensure that the array size remains constant for each of the output and interconnect topologies investigated, and to provide sufficient PALUs to make use of the hierarchical interconnect employed. Results from section 6.3.5 show than on average only 78% of PALUs in the FPGA-based filter were utilised. It is therefore prudent to assume that if the PLA is to produce competitive PALU utilisation, then the same number of PALUs should be

used; this would require a lower PALU utilisation in the PLA of around 50%.

All other GA and filter parameters are the same as that detailed in section 6.3.3. Therefore for a 11x9 PLA-based filter implementing *Route 4* interconnect and *Output 3* tap sequence, 1449 bits are required to encoded the configuration string. This translates to a maximum search space of 2^{1449} possible bit string configurations, which must then be successfully manipulated by the genetic algorithm.

6.4.4 Investigation of Genetic Operator Parameters

The configuration bit string used to program the PLA-based filter is compartmentalised and encoded in a different way to the bit string used to configure the FPGA. In addition, the architectural differences between the two programmable platforms are significant enough to produce search spaces with very different fitness landscapes. These two factors potentially effect the suitability of the original crossover and mutation parameters used by the GA to manipulate the FPGAs configuration. As a result the same investigations detailed in section 6.3.4 will again be performed, this time to determine the effects of crossover and mutation when using the GA to configure the PLA for coefficient generation. Crossover probabilities of 0 and 60% are therefore examined, in addition to mutation rates of $1/N$, $2/N$ and $3/N$. Three of the four interconnect topologies were implemented for each crossover and mutation parameter investigated. *Output 1* was arbitrarily selected as the fixed output topology for each investigation. Tables 6.4 and 6.5 display the average fitness of the coefficient sets generated for each PLA interconnect sequence as a results of varying crossover and mutation probabilities respectively. Despite the differ-

Connection Topology	Average Fitness with Crossover	Average Fitness without Crossover
Route 1	8.5387 = 94.9%	8.6018 = 95.6%
Route 2	8.3010 = 98.1%	8.7930 = 97.7%
Route 3	8.8757 = 98.6%	8.8824 = 98.7%

Table 6.4: *Performance of PLA connection topologies in generating 31-tap low-pass filter configured using genetic algorithm with and without crossover.*

ences in architecture and bit string encoding, both the FPGA and PLA-based filters perform equally well without crossover as they do when it is employed. This is for the same reasons of epistasis discussed in section 6.3.4. Whilst the mutation rate of $2/N$ produced coefficients of slightly better fitness (between 0.3 and 1.1% depending on the PLA interconnect topology using

Connection Topology	Average Fitness at $P_m = 2/N$	Average Fitness at $P_m = 3/N$
Route 1	8.6450 = 96.1%	8.5770 = 95.3%
Route 2	8.8560 = 98.4%	8.8213 = 98.0%
Route 3	8.9192 = 99.1%	8.8712 = 98.6%

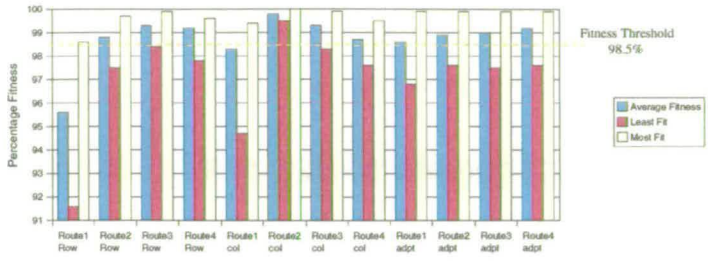
Table 6.5: *Performance of PLA connection topologies in generating 31-tap low-pass filter configured using genetic algorithm with variable mutation rates and no crossover employed.*

a mutation rate of $1/N$), it was decided to keep the original parameters used for the FPGA by maintaining both the original mutation rate at $P_m = 1/N$ and removing crossover. This would also provide more accurate future comparisons between the two programmable platforms.

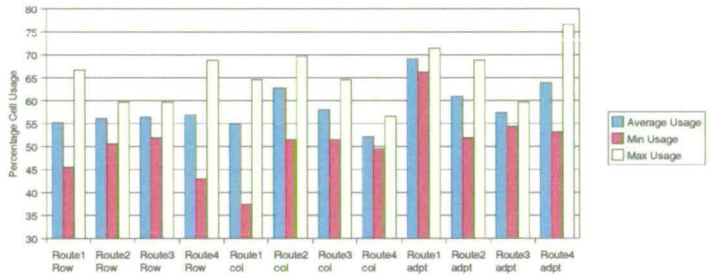
6.4.5 Performance Comparison of PLA Topologies

The performance of each PLA routing and output topology was based on the same four performance criteria used to investigate the FPGA outlined at the beginning of section 6.3.3. For completeness the criteria a listed here again; and includes the fitness of the filter, the number of PALUs used, the degree of PALU re-use (generation of partial products), and the total number of shift, add and subtract operations required to implement the specified set of coefficients. The results of each criteria, averaged over the ten investigations for each PLA topology, are shown in Figure 6.16.

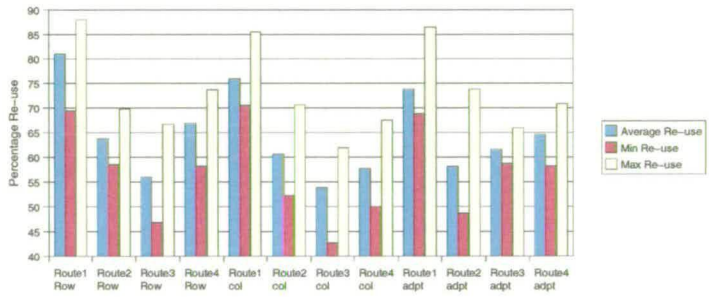
Coefficient Fitness: Both output topologies 1 and 3 produce coefficients of greater fitness the higher the degree of routing available between PALUs. This is not the case when *Output 2* is used, in fact higher interconnectivity reduces the ability of the GA to find suitable PLA configurations when column-based tap outputs are employed. However, A PLA combination of *Output 2* and *Route 2* provides the genetic algorithm with a programmable architecture which consistently produces highly fit filter coefficients, and was the only topology to produce a set of filter coefficient which exactly matched those presented in table 6.1. Remember that a fitness of $\geq 98.5\%$ was required to produce a transfer function with acceptable low-pass characteristics. At least one in ten evolutionary runs resulted in an acceptable filter response for each of the PLA output and interconnect topologies examined; this is a considerable improvement over the FPGA platform. In addition, the *Output 2*, *Route 2* PLA topology was the only programmable architecture in which all ten configurations generated by the GA produced acceptable



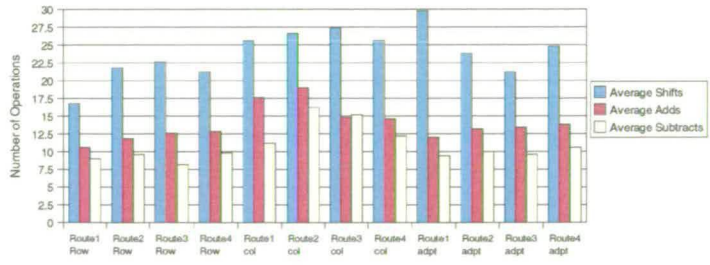
(a) Success of FIR filter based on fitness criteria.



(b) PALU usage (coverage) required by FIR filter.



(c) PALU re-use exploited by FIR filter.



(d) Operations performed by PALU to implement FIR filter.

Figure 6.16: *Performance of PLA topologies to autonomously generate a 31-tap low-pass FIR filter.*

coefficients.

PALU Utilisation: On average, the greater the number of PALU's used within a PLA topology the better the fitness of the filter coefficients produced. This is particularly true for PLA topologies employing *Output 2*, where the degree PALU utilisation closely mirrors coefficient fitness. Approximately 57% of PALUs were required to implemented the desired coefficient set. In real terms this equates to an average increase of 6 PALUs per implementation of the low-pass filter coefficient set, when compared to the average number of PALUs utilised on the FPGA. This result demonstrates that although the array dimensions of the PLA are larger than the FPGA, both platforms utilise roughly the same number of PALUs. The PLA therefore remains competitive with the FPGA in terms of PALU utilisation, however, the quality of coefficient set generated on the PLA is markedly higher than that produced on by the FPGA.

PALU Re-use: *Route 1* (local routing) promotes the highest degree of PALU re-use, independent of the output topology. This is again due to the high level of dependence between PALUs inherent in the *Route 1* interconnect topology, a relationship which was also found with the FPGA architecture. However, unlike the FPGA architecture the critical path through the PLA is not influenced by interconnect topology because of its 2D column-based structure, and therefore cannot be used as a direct measure of linkage between PALUs. Also, the *ideal connection topology* between columns of PALU means that the available connectivity between neighbouring PALUs is considerably higher than that found on the FPGA. As a result PALU re-use on the PLA platform is three times lower than that on the FPGA.

Route 1 also produces a poorer quality of filter coefficients than when other routing topologies are used. The genetic algorithm therefore demonstrates that *Route 1* connectivity is the least effective means of generating the specified set of coefficients. This stems directly from the topologies flat interconnect hierarchy which does not provide direct routing between non-adjacent columns of PALU.

PALU Operations: Almost identical trends in the use of PALU operations can be seen between both the PLA and FPGA architectures. In all PLA topologies the number of shift operations utilised is approximately twice that of addition or subtraction; and the number of PALU additions is consistently greater than the number of subtractions. Again, these characteristics are particularly desirable for efficient partial product generation using multiplierless design techniques such as CSD and POF. Remember that the genetic algorithm has no a priori knowledge

of POF or either of the programmable platform architectures. The GA has therefore provided a strong indication as to the natural suitability of both programmable platforms for POF based FIR Filter coefficient generation.

6.4.6 Graphical Representation of PLA-Based FIR Filter

The graphical C program detailed in section 6.3.6 was modified to accommodate the various PLA topologies. Four postscript templates were developed to reflect each of the four interconnect topologies investigated; these are shown in Appendix B. The hierarchical interconnect of each PLA topology is colour encoded. Local, or nearest neighbour connectivity, is denoted in yellow, level-2 interconnect in green, level-4 in red, and column based connectivity in light blue (the same colours as the PALU). Connections programmed to logic '0' are shown in pink, and those connected to the filter input, $X(n)$ are displayed in dark blue. Coefficient tap outputs are labelled and also highlighted. As with the FPGA graphical program, PALU operations are denoted by their relevant signs. Figure 6.17 displays the PLA configuration of the best coefficient set generated by the GA. As mentioned above it was implemented on a PLA with *Route 2* interconnect and *Output 2* tap placement.

6.5 Comparison of PLA and FPGA-Based Filter Platforms

The genetic algorithm has provided a means of independently appraising the suitability of both the FPGA and PLA-based filter platforms for implementing the coefficient set used to describe the benchmark 31-tap low-pass FIR filter presented in this chapter. Whilst comparisons have already been made in section 6.4.5, this section identifies those critical comparisons which remain.

The results presented show that the average quality of coefficients produced by the GA on the PLA based FIR-filter (98.6%) significantly outperform the fitness of those produced on the FPGA architecture (94.6%). It is therefore apparent that the PLA-based filter consistently produces coefficients sets which fulfill the desired low-pass specification ($\geq$ 98.5%), and as such is more suited to FIR filter coefficient generation than the FPGA-based approach. This can be further substantiated by recalling that a deviation in the accumulated fitness of a coefficient set by more than 0.5 to 1.0% can have significant impact on the performance of the transfer function produced. An average drop of 4% in the fitness of coefficients produced on the FPGA

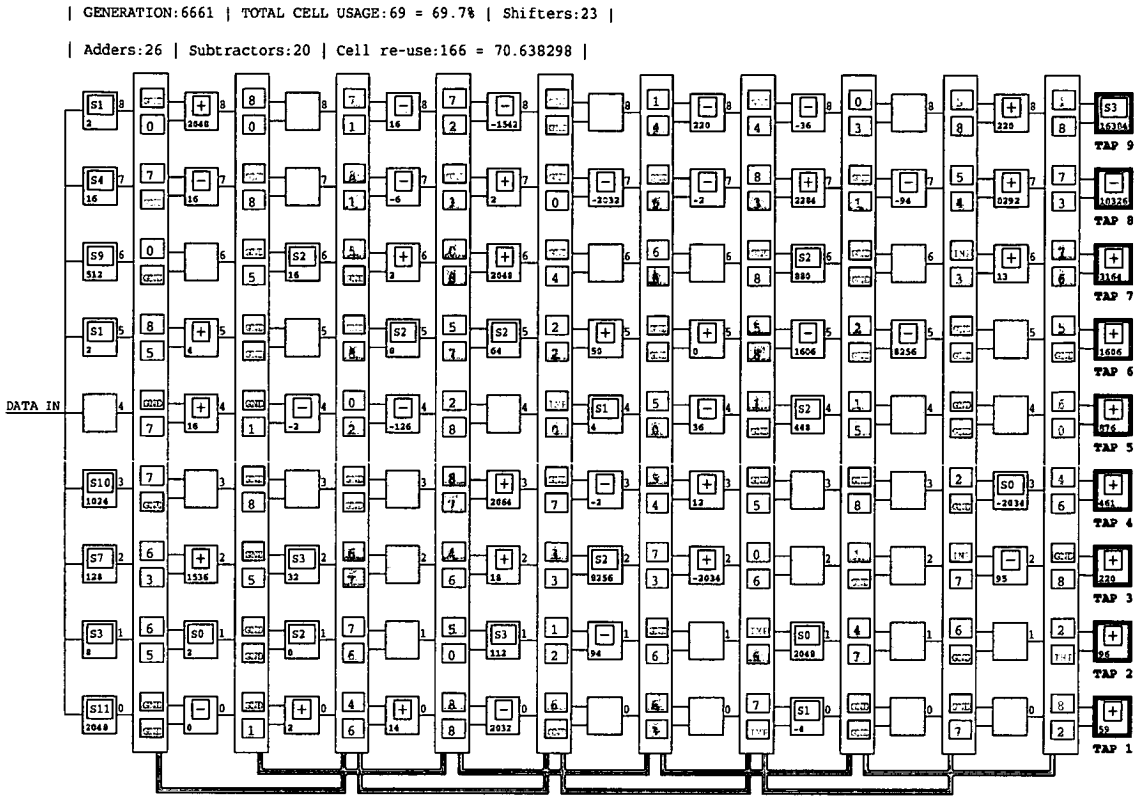


Figure 6.17: *Example PLA configuration of 31-tap low-pass filter filter.*

is therefore a significant indication as to the superiority of the PLA-based architecture.

Results also indicate that in both programmable platforms a strong relationship exists between the degree of linkage, or freedom of connectivity, provided by interconnect topologies between PALUs, and the amount of PALU re-use within each array. It has been shown that the higher the degree of linkage (dependency) between neighbouring PALUs the greater the degree of re-use. This relationship ship can be gauged by the critical path that is created by each interconnect sequence, and the availability of shorter routes which might be taken along the critical path. Routing topologies which display the highest degree of linkage and least freedom of interconnect along the critical path are the AFFA and *Route 1* interconnect sequences for the FPGA and PLA respectively.

The following summarises the PLA and FPGA topologies best and least suited to autonomously implementing the benchmark FIR filter coefficient set using EHW.

- The best average coefficient fitness produced on the FPGA was achieved using the LSIS-

CFFLA-BLOS topology at 97.3%

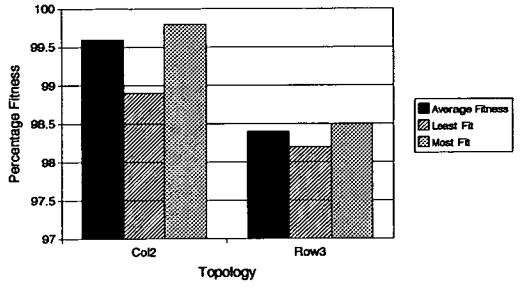
- The worst average coefficient fitness produced on the FPGA was achieved using the BLIS-AFFA-AOOS topology at 89.9%
- The best average coefficient fitness produced on the PLA was achieved using *Route 2* and *Output 2* at 99.8%. For the purposes of further investigation this PLA topology will now be denoted *Col2*.
- The worst average coefficient fitness produced on the PLA was achieved using *Route 1* and *Output 1* at 95.6%

The second most effective programmable topology was again achieved using the PLA, in this case with *Route 3* interconnect and the column-based tap output sequence, *Output 1*. For the purpose of further investigation this PLA topology will now be termed *Row3*.

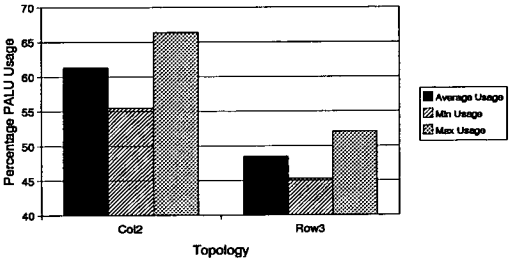
6.5.1 Further Investigations

In order to further validate the results obtained through configuration of the low pass filter, the two most effective programmable topologies, *Col2* and *Row3*, were analysed. A second filter specification was chosen to be the *20-tap Hilbert transformer*, designed using the Remez exchange algorithm developed by McClellan et. al. and benchmarked in [130]. This filter was chosen as it required 10 taps to implement in folded form and is therefore of similar length to the 9 distinct taps required to implement the 31-tap low-pass filter benchmarked previously. However, the Hilbert transformer has a different coefficient distribution which will test the general suitability of both PLA architectures, which have very different output topologies. Whereas the low-pass filter response requires a set of coefficients who's magnitudes increase with tap length, the Hilbert transformer coefficient distribution varies in magnitude along the length of the filter. Finally the Hilbert transformer is also represented using a 16-bit 2's compliment encoding, which again matches the PLA specification required for the low-pass filter.

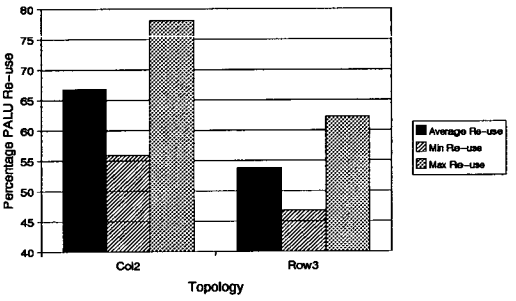
Col2 was implemented using 11 columns and 10 rows. *Row3* implemented the Hilbert transform using 13 columns and 9 rows. Both topologies therefore have a comparable number of PALU's. The same experimental setup was used as for the low-pass filter investigation. Results are shown in Figure 6.18.



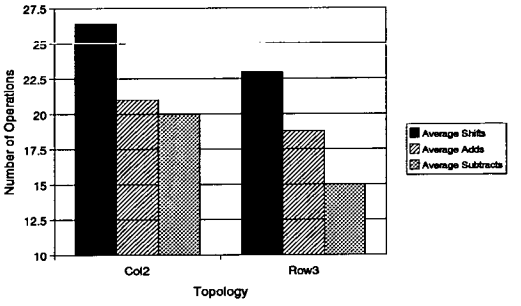
(a) Success of FIR filter based on fitness criteria.



(b) PALU usage (coverage) required by FIR filter.



(c) PALU re-use exploited by FIR filter.



(d) Operations performed by PALU to implement FIR filter.

Figure 6.18: *Performance of* Col2 *and* Row3 *PLA topologies to autonomously generate a 20-tap Hilbert transform FIR filter.*

Col2 enables the genetic algorithm to produce coefficients for the Hilbert transform with considerably better fitness than those generated using *Row3*. Comparison between coefficient fitness and PALU usage (Figure 6.18(a) and Figure 6.18(b)) further demonstrates the relationship between PALU usage and coefficient fitness. *Col2* therefore utilises approximately 15% more PALU's than *Row3*, achieving greater filter performance.

Reproduction of the desired filter response inside each PLA has been of primary importance. Filter performance on the PLA topology with column-based tap placement (*Output 2*) is not dependent on coefficient distribution. Row-based tap placement (*Output 1*) performs better on filters who's coefficients are distributed in ascending order. This observation outlines a restriction in the PLA architecture as partial products for each coefficient are generated column-by-column. Each PALU therefore relies on terms generated by previous columns to produce the relevant product. *Output 1* is therefore detrimental to the effectiveness of the basic PLA architecture for non-ordered coefficient filters. This is because the magnitude of partial products

generated in each proceeding column may not necessarily increase. This can be overcome using *Output 2*, or has been shown in section 6.4.5, by increasing the degree of available interconnect between PALUs.

6.6 Summary

This chapter has presented the development, and evaluation of two programmable platforms tailored for implementing FIR filter coefficient multiplication using EHW. Coefficient sets are implemented on either an FPGA or PLA-based programmable architecture, both of which replace explicit coefficient multiplication with a distributed series of bit-shifts, additions and subtractions. Each programmable platform employs an embedded genetic algorithm designed to autonomously configure the PLD for a given filter specification. The genetic algorithm was used to investigate the most suitable programmable architecture for implementing high-performance multiplierless digital filters, and provided parameterisation of the key genetic operators: crossover and mutation. Initial tests however have shown the limitations of crossover as an effective means of generating a specified coefficient set on either programmable platform.

A 31-tap low-pass FIR filter was benchmarked to enable comparisons between the performance of the PLA and FPGA architectures. Each architecture was implemented using a number of filter input, tap output and PALU interconnect topologies. The performance of each topology was evaluated based on the coefficient fitness, area utilisation, and PALU re-usability of the configurations generated by the genetic algorithm. Coefficient fitness is the most important measure of FPGA/PLA performance. Results demonstrate that the PLA-based architecture considerably outperformed the FPGA in terms of the quality of the coefficient sets produced. Investigations show that *Col2* produced filter coefficients of higher fitness than other topologies, when autonomously configured using the genetic algorithm. On average the PLA produces coefficient sets with a fitness score 4% higher than the FPGA. The PLAs dominance over the FPGA is attributed to the higher degree of flexibility afforded by the PLA interconnect topologies which utilise a hierarchical connectivity; and the fact that the critical path of the PLA is markedly shorter than the FPGA. Both these factors have been shown to effect the degree of interdependence (linkage) between neighbouring PALUs. Greater flexibility of interconnect and short critical paths therefore reduce PALU linkage and increase FPGA/PLA performance. *Col2* was also shown to be the most flexible PLA architecture for implementing filters with a non-uniform coefficient distribution, significantly out-performing the next best programmable

topology (*row3*).

Whilst the *Col2* PLA architecture has been shown to be the most effective in generating FIR filter coefficients using EHW, its currents implementation using an ideal interconnect between columns of PALU is unrealistic and would require prohibitive routing in VLSI as the number of PALU columns and rows increases to match filter complexity. Chapter 7 therefore investigates the translation of the *Col2* architecture into a synthesisable VHDL netlist for implementation in silicon. Physical constraints will be examined, such as timing along the critical path and the degree of interconnect required to implement a functionally acceptable *Col2* architecture without ideal interconnect.

Chapter 7
Translating the *Col2* PLA Topology into Hardware

7.1 Introduction

All of the programmable platforms currently investigated have used software simulation to evaluate the performance of the filter coefficient sets configured on them by the genetic algorithm. The performance evaluation of both the PLA and FPGA-based FIR filters therefore exhibit extrinsic evolution in EHW terms, as discussed in Chapter 2. In order for faster and more realistic generation of filter coefficients to occur, the programmable platform on which the filter is to be implemented must be realised in physical hardware. Intrinsic real-time evaluation of the PLA architecture configured to implement a given coefficient set is then possible.

Analysis of both the FPGA and PLA-based EHW platforms investigated in Chapter 6 have shown that a PLA architecture capable of column-based tap placement with localised and 2-level PALU interconnect, *Col2*, is the most effective programmable topology for implementing an FIR coefficient multiplication unit using EHW. However, its *ideal interconnect topology* does not make it suitable for implementation in hardware, which is of little use to real world SoC signal processing applications. In addition, FIR filters are crucial for robust data communication and manipulation. DSP devices are frequently employed in environments where issues such high processing speed, low physical area, and device reliability are highly critical, such as in space applications. For many such applications DSPs must maintain functionality over prolonged periods in harsh environments. Built in reliability of FIR filter devices is therefore required. The performance and sustained reliability of hardwired FIR filters is therefore of great importance.

This Chapter presents the translation of the *Col2* PLA architecture from an RTL-level behavioural VHDL model into a physically realisable, technology specific netlist using *Synopsys Design Analyser* synthesis software and Alcatel's 0.35 μm MTC45000 technology library. The architectural limitations of the *Col2* PLA are identified in order to develop a physically realistic PLA structure. The synthesised PLA netlist is then compared with the original *Col2* PLA

architecture using the GA. Finally the real-world performance of the netlisted PLA is further investigated by examining the ability of both the GA and the PLA to adapt to an increasingly high number of faults randomly introduced onto PALUs in the PLA architecture.

7.2 Synthesis and Performance Analysis of PLA-Based Filter

Translating *Col2* into a synthesisable IP core requires modifications to be made to the original array of *ideal interconnects* between PALUs, detailed in section 6.4.1. The ideal connectivity model is not suitable for hardware implementation as it incurs a large area for both routing and control logic, due to an excessive interconnect. This would result in long delays between PALU's, and high capacitive loads from excessive fanout on interconnect pins. In order to minimise these problems the interconnect array was modified such that each PALU input could route to one of only three PALU's from the previous column. So as to maximise connectivity between columns, no two inputs were routed to the same set of three PALU's. The reduced connectivity architecture centres around the position of the current PALU in the column. *Ain* is connected to one PALU above, below and including that of the PALU at the same location in the previous column. *Bin* then connects to the second, third and fourth PALU directly above that of the PALU in the previous column. Figure 7.1 displays an example of the reduced connectivity model.

The amount of control logic for the interconnect array is therefore reduced, along with interconnect area, signal delay, and drive-strength. Also, this complexity does not increase with column height (as would be the case with the ideal interconnect). This ensures that the PLA architecture remains scalable.

7.2.1 Comparative analysis with RTL 'ideal' model

In order to determine if the reduced connectivity PLA architecture performed as well as the ideal interconnect, *Col2* was modified and simulated using the same benchmark low-pass filter detailed in section 6.2 of Chapter 6, and the same experimental setup as that used for the original *Col2* PLA presented in sections 6.2.1 and 6.4.3. The modified *Col2* PLA is named *Col2_reduced* for clarity and was again written in VHDL at the RTL level. A second investigation of *Col2_reduced* was also performed to determine the effects of changing PLA dimensions. Column width was reduced from 11 to 6, and the number of rows was instead extended to 16.

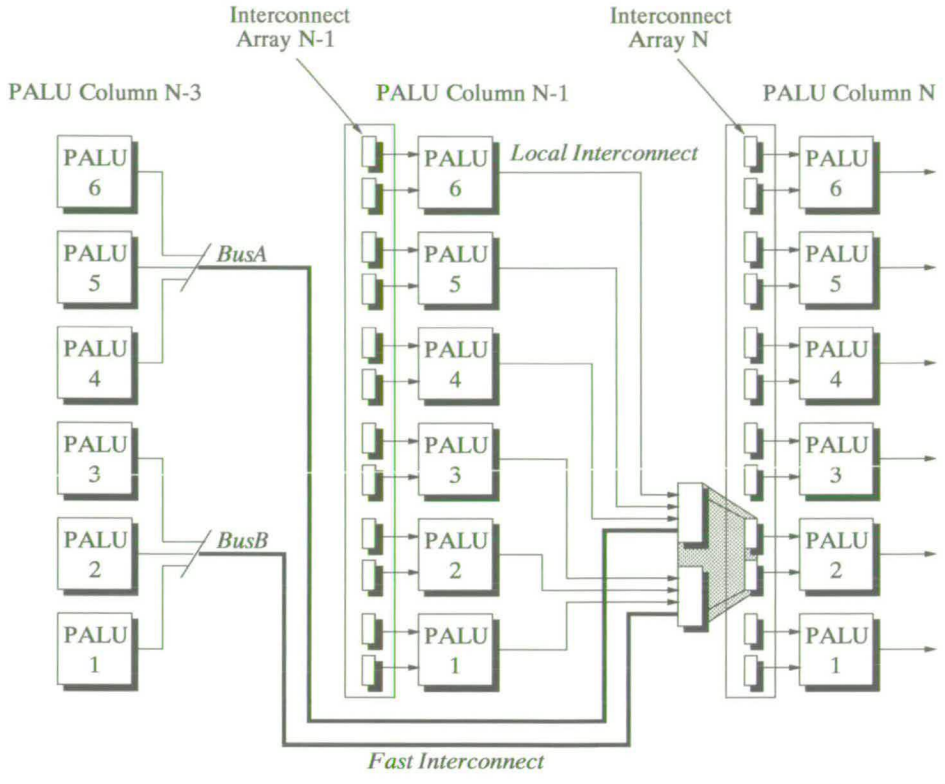
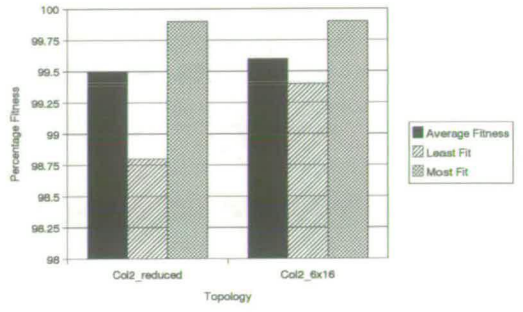


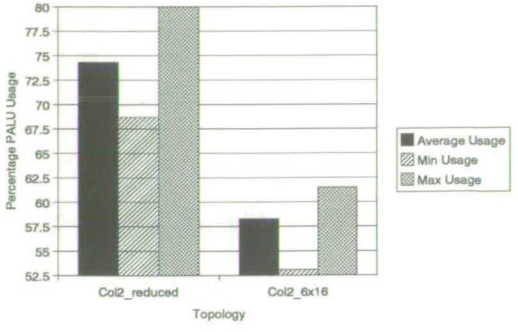
Figure 7.1: *Example of reduced connectivity between PALUs*

In this case the bottom 9 PALUs in the final column were connected to output taps. These dimensions were designed to roughly maintain the number of PALUs within the PLA, whilst reducing the latency of the circuit. The same experimental setup was again employed, and the PLA was identified as *Col2_6x16*. Figure 7.2 displays averaged data of the results obtained.

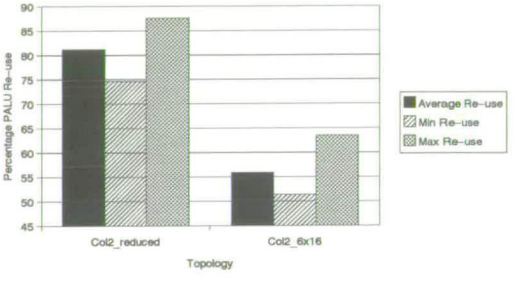
The average fitness of coefficients generated using *Col2_reduced* is almost identical to those produced using the original *Col2* topology. However *Col2_reduced* used approximately 15% more PALUs than *Col2*, and re-used on average 81%; around 20% more than the original *Col2* PLA. This supports evidence presented in Figure 6.16 which links reduced connectivity with greater PALU re-use and in some cases poorer filter performance. The fact that more PALUs are required to implement filters of high quality suggests that *Col2_reduced* still provides the genetic algorithm with a means of counter-acting the negative effects of reduced connectivity between PALUs. Altering dimensions of the PLA in *Col2_6x16* maintained the quality of the filter coefficients, but greatly reduced both the number of PALUs used and the degree of re-use when compared to *Col2_limited*. In fact usage and re-use are shown to be comparable to or lower than those produced using the original *Col2* PLA. The results highlight the flexibility of



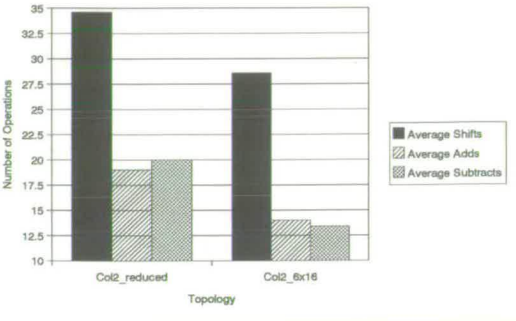
(a) Success of FIR filter based on fitness criteria.



(b) PALU usage (coverage) required by FIR filter.



(c) PALU re-use exploited by FIR filter.



(d) Operations performed by PALU to implement FIR filter.

Figure 7.2: *Performance of* Col2_reduced *and* Col2_6x16 *PLA topologies to autonomously generate a 31-tap low-pass FIR filter.*

the PLA to adapt to varying dimensions and maintain filter performance. Ratios between shift, addition and subtraction operations remain similar to those originally identified in Chapter 6 throughout.

7.2.2 Synthesis Details

Due to the success of the *Col2_6x16* topology, which was written in VHDL at the RTL level, a *PLA core* with 6 columns and 5 rows was synthesised using the Alcatel MTC45000 library. Five rows were chosen to maintain a compact PLA core that could readily be synthesised. Multiple cores are simply connected during initial parameterisation of the PLA-based FIR filter platform. Therefore a PLA can be sized according to the maximum number of taps required for a specific range of applications. A total of $C + 1$ clock cycles are required to multiply input data with the desired coefficient set, where $C = 6$ and is the number of PALU columns. As a result,

the throughput of the PLA is *not* effected by filter length. This is a considerable advantage over single multiplier MAC filter architectures. Top-down synthesis was performed using ***Synopsys Design Analyzer***. This was shown to produce better timing and area results than synthesis using a bottom-up approach. Appendix C.1 displays the synthesis script used. The scalability of the PLA core was examined by synthesising it at six operational clock frequencies: 10MHz, 25MHz, 50MHz, 80MHz, 90MHz, and 100 MHz; all PLA data-widths were set at 16-bits. Theses frequencies were chosen to reflect typical timing constraints required on high speed SoC bus architectures, which range from 60 to 100MHz. Result can be seen in Figure 7.3. Area remains relatively constant from 10 to 50 MHz. Between 50 and 100MHz the area of the synthesised PLA core increases approximately linearly. For technologies smaller than 0.35μm, faster throughput could be achieved.

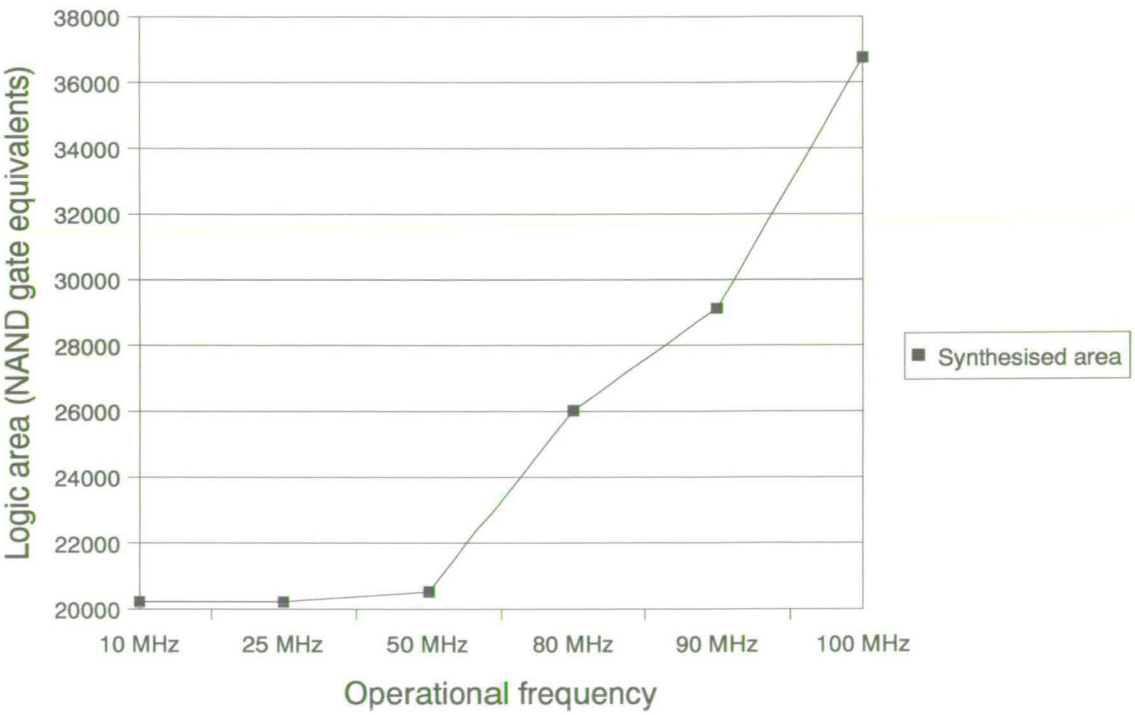


Figure 7.3: *Logic area of PLA core as a result of synthesis for increasing operational speeds.*

The critical timing path was found to lie between the *BusA/BusB* input of any given set of interconnect logic, and the output register of the PALU which is associated with this interconnect. This is explained in more detail in Figure 7.4. The largest delay is incurred through the adder/subtracter unit of the PALU. One way to reduce this would be to customise the ad-

der/subtracter block for a specific silicon technology. This would limit the general portability of the *PLA core*, but further increase its performance.

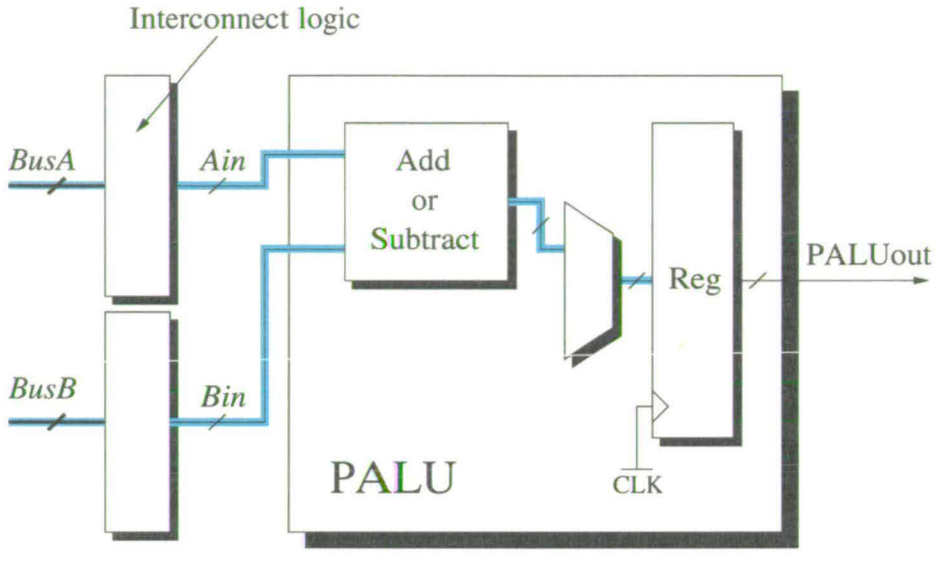


Figure 7.4: *Critical delay path through PLA architecture.*

Figure 7.5 displays the Leapfrog VHDL simulation of the 6x5 *PLA Core* netlist synthesised to operate at 10MHz and then back annotated into the simulation testbench presented in Appendix C.2.

The *PLA Core* presented has been programmed to multiply $X(n)$ by the coefficient set $\{1, 7, 16, 21, 33\}$, representing taps 1 to 5 respectively, defined as *Output_Port(i)* in the waveform of Figure 7.5. The coefficients were configured on the *PLA Core* using the bit-string displayed in *Memory_Contents*. The waveform signal *Pla_Data_Stream* is simply the binary data held in *Memory_Contents* as it is fed bit-serially into the PLAs serial-to-parallel shift register as shown in Figure 6.12 of Chapter 6. Ten input stimuli (16-bit words) representing the filter input, $X(n)$, were applied to the *PLA Core* via *PLA_Signalinput*. Each of the ten input vectors in the set $\{1, 25, 49, 385, 553, 271, 1, 449, 107\}$ is fed in turn to the *PLA Core* which then takes 7 clock cycles to perform the distributed coefficient multiplication before the result is present on *Output_Port()*. The 7 cycle latency between *PLA_Signalinput* and *Output_Port()* can clearly be seen by the red markers in Figure 7.5, which indicates when the *PLA Core* has finished processing the current coefficient multiplication. For example, the third input word, 49, when multiplied by the 5-tap coefficient set can be seen to display the correct corresponding tap outputs 49, 343, 784, 1029 and 1617.

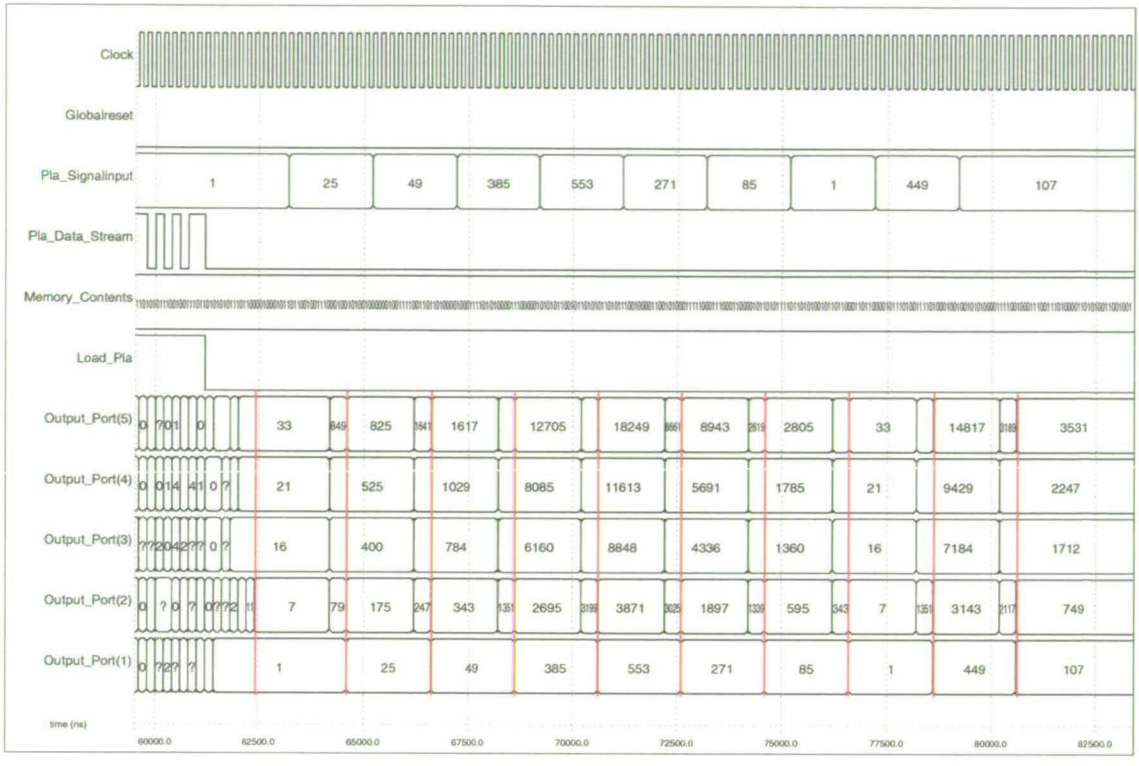


Figure 7.5: *Simulation waveform of 6x5* PLA Core *VHDL netlist synthesised at 10MHz.*

7.3 Fault Tolerant Characteristics of PLA-Based EHW Platform

Figure 7.2(b) shows that on average around 42% of the *Col2_6x16* PLA topology remains redundant after the low-pass filter is implemented. This provides the GA with sufficient PALU resources to reconfigure the PLA if sections of the architecture become damaged. The PLA-based FIR filter platform therefore exhibits fault tolerance through controlled redundancy. Karri has already shown this fault-tolerant method to be efficient in [112].

Fault tolerance systems are widely used in space applications such as commercial satellite communication where hardware deteriorates due to damaged caused by cosmic rays, and in other inhospitable environments where human intervention is difficult or impossible. Systems must therefore maintain functionality despite factors such as severe temperature variation, radiation and operational ware. Conventional fault tolerant VLSI systems employ techniques such as *check-pointing* [110], *concurrent error detection* [111] and *redundancy*. There purpose is to maintain system operation, or prevent further successive faults by minimising the damaged sustained [18]. However, fault tolerant systems are costly as they reduce operational speed and

increase physical area.

Alternative approaches to the design of fault tolerant systems have recently been proposed using evolvable hardware [21, 131–133]. Such approaches provide novel techniques for fault recovery and prevention without the need of additional redundancy, fault detection or diagnosis. Instead a genetic algorithm is used to monitor system performance and reconfigure aspects of a circuit to counter-act any deleterious faults. Fault tolerant systems which employ EHW must therefore be able to adapt on-line when required. Hardwired GAs are capable of running considerably faster than those implemented in software on general purpose micro-computers, and are therefore suitable for applications which require online adaptation. As a result, GAs are frequently mapped onto Programmable logic devices (PLDs) so that the fitness function can later be modified for different design criteria [17, 121–123]. Custom EAs have also been implemented on ASICs [124, 125]. In such cases the fitness algorithm is then set for a specific application.

For EHW to provide a competitive solution to conventional fault tolerant design, EHW resources must be smaller than those required by conventional fault tolerant architectures. The benefits of using a single fixed EHW resource become more apparent as circuit size or complexity increases. Inversely, hardware requirements for conventional fault tolerant designs will continue to grow.

7.3.1 Introducing Faults into the PLA-Based FIR Filter

The *Col2_6x16* PLA architecture (RTL description) was subjected to four increasingly large numbers of faulty PALUs. These faults covered 0%, 5%, 13% and 25% of the PLA architecture. Each individual fault was obtained by pulling both inputs of a given PLA to zero and setting it to shift-by-zero, effectively simulating a "Stuck at zero" fault. This was achieved by "freezing" sections of configuration string which related to the selected faulty PALUs. For each increasing percentage of faults the low-pass filter coefficients were evolved ten times on the PLA using the genetic algorithm. Again, ten randomly generated populations of configuration-strings were created for each of the ten filter coefficient sets evolved. The dimensions of *Col2_6x16* were maintained to provide limited redundancy for when faults were introduced into the PLA.

Faults were placed at random for each level of coverage. The same faults were then maintained over the ten times the low pass filter coefficients were evolved so to obtain an average.

Figure 7.6 displays the topology of PALU faults for each level of fault coverage.

7.3.2 Analysis

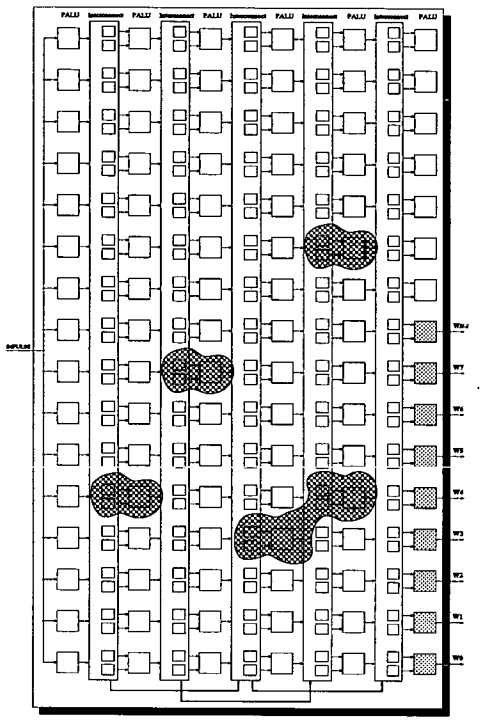
The ability of the fault tolerant hardware platform to adapt to or sustain increasing faults was investigated through the same four criteria identified in Chapter 6: The fitness of the filter evolved, the number of PALUs used, the degree of PALU re-use (generation of partial products), and the total number of shift, add and subtract operations required to implement the desired coefficients. Results are shown in Figure 7.7.

Figure 7.7 reveals that at PALU faults from 0 to 13%, the genetic algorithm is in each case able to evolve a filter with a maximal fitness of 99.9%. When the PLA is 25% faulty a 0.5% decrease in the fittest filter solution is incurred. The average fitness of filter coefficients evolved remain above 98.5% until 25% of the PLA experiences faults. Recall from Chapter 6 that a fitness $\geq$ 98.5% was required to produce a transfer function with acceptable low-pass characteristics and a gain no less than -52 dB. An example of a typically acceptable response for the low-pass filter is shown by the green transfer function in Figure 6.1. Variation of the least fit filters evolved by the GA is more marked as the percentage of faults in the PLA increases.

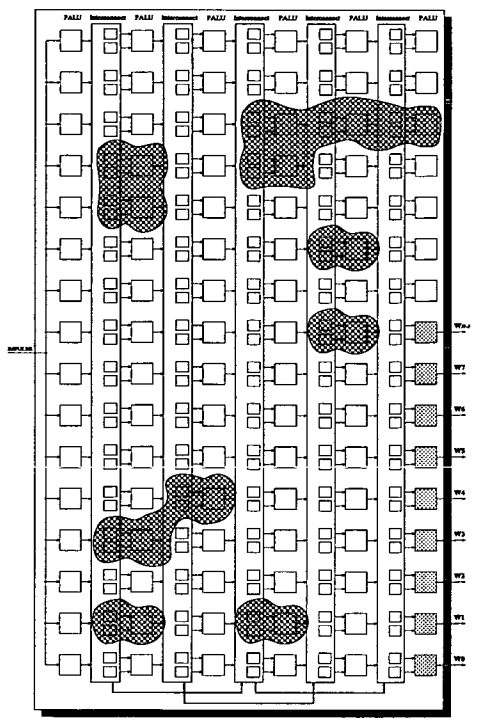
The average number of PALUs used to implement the low pass filter reduces as the percentage of faults found in the PLA increases. However, only a 10% reduction in PALU usage is experienced on the PLA despite a 25% decrease in the number of functional PALUs available. This suggests that the PLA provides the GA with a means of counter-acting the deleterious effects of PALUs which are "stuck at zero". Comparisons between figures 7.7(a) and 7.7(b) reveal that as fewer PALU resources are made available to the GA through faults, a reduction in filter performance is experienced. The number of PALUs required to implement the filter therefore relates directly to the fitness of the evolved solution.

Accounting for variations of faults at 5 and 13% of the PLA area, the average number of PALUs reused within the PLA (to generate partial product terms) remains relatively constant, with an overall reduction of less than 5%. Again this supports the notion of a robust PLA architecture capable of providing an adaptable environment for fault tolerant design using EHW.

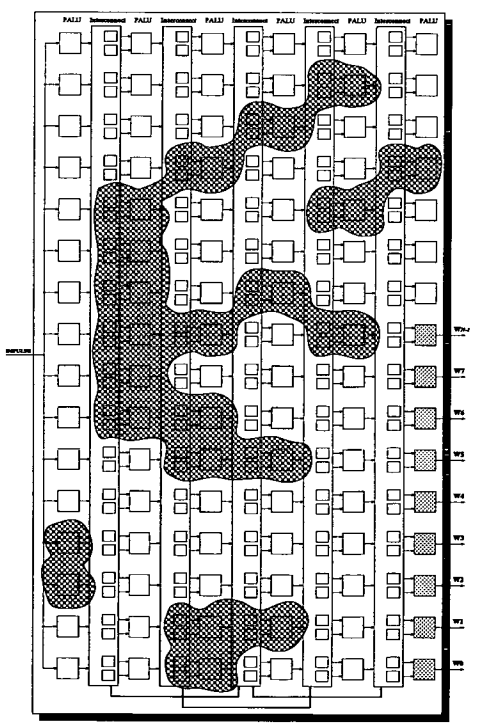
Regardless the degree of faulty PALUs, the number of shifter operations selected by the GA is double that of either additions or subtractions. As expressed in section 6.3.5 and 6.4.5 in Chapter 6, this is encouraging as programmable shifts consume considerably less power then



(a) 5% PALU faults.

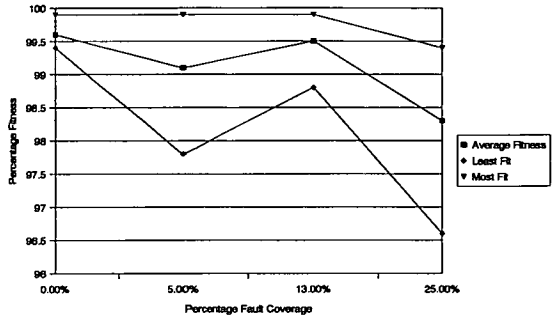


(b) 13% PALU faults.

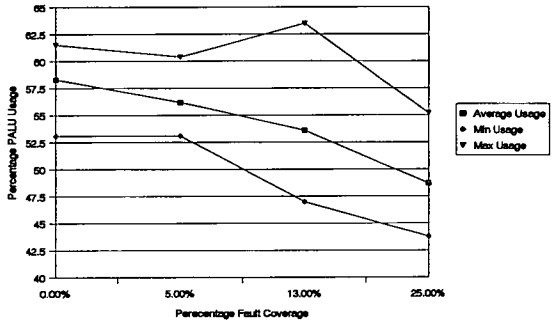


(c) 25% PALU faults.

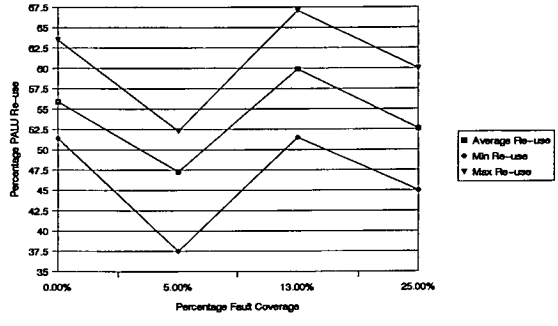
Figure 7.6: *"Stuck-at-Zero" fault topologies covering PLA*



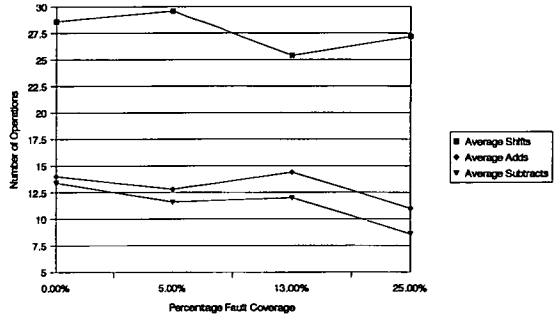
(a) Fitness of FIR filter coefficients based on increased percentage of faults in PLA.



(b) Number of PALUs utilised to generate desired FIR filter coefficients based on increased percentage of faults in PLA.



(c) Amount of PALU re-use based on increased percentage of faults in PLA.



(d) Type of PALU operation based on increased percentage of faults in PLA.

Figure 7.7: *Analysis of* Col2_6x16 *PLA architecture with increasing percentages of faulty PALUs*

either addition or subtraction operations, and are considered to be the primary means of product generation when using multiplierless filter techniques such as POF design.

Of course the placement of faulty PALUs will greatly effect the ability of the GA to evolve high quality filters. Faults close to, or directly on PALUs which are connected to coefficient taps will have a more detrimental effect than those distributed in the centre of the PLA. This accounts for the poorer average fitness performance of filters generated on the PLA with a faulty area of 5% compared to those generate by the GA on a PLA with 13% faults. Given the added number of redundant rows present in the PLA architecture it would be possible to adapt the platform such that taps could be moved to other PALUs which are not near faults, or are themselves faulty, but still located on the final column. An example PLA configuration of the 31-tap low pass filter evolved with 99.9% correctness and 13% of its PALUs faulty is illustrated in Figure 7.8. Faulty PALUs are shown as dark green anomalies.

7.3.3 Population Initialisation After Fault Detection

In each of the fault scenarios investigated in section 7.3.1, populations of configuration-strings were randomly generated. However, two other approaches to population initialisation are available. In this Chapter they are termed *Population seeding*, and *Population recall.* Both of these approaches were investigated to see if fault recovery times after detection could be decreased when compared to random population initialisation. The method of population seeding applied in this Chapter involved taking the fittest solution stored from the previous evolutionary run (and currently in operation on the PLA), and placing it into a population of 99 randomly generated configuration-strings. Population recall simply involves re-introducing the most recent population of configuration-strings evolved , and using this as the initial start point.

The effectiveness of both approaches was examined using the same *Col2_6x16* PLA topology with 13% of its PALUs "stuck-at-zero", as shown in Figure 7.6(b). So as to obtain an averaged performance the same low-pass filter was evolved ten times using the population seeding approach. In each case the PLA configuration shown in Figure 7.8 was used as the seed. All other configuration-strings were randomly generated. A randomly selected final population, evolved for the low-pass filter with no faults in the PLA, was used as the initial population for the recall approach. As no configuration-string needed to be randomly generated this scenario was run only once. The performance of each initialisation approach was determined to be the number of generations required by the GA to produce a filter response with a fitness $\geq$ 98.5%. Figure 7.9

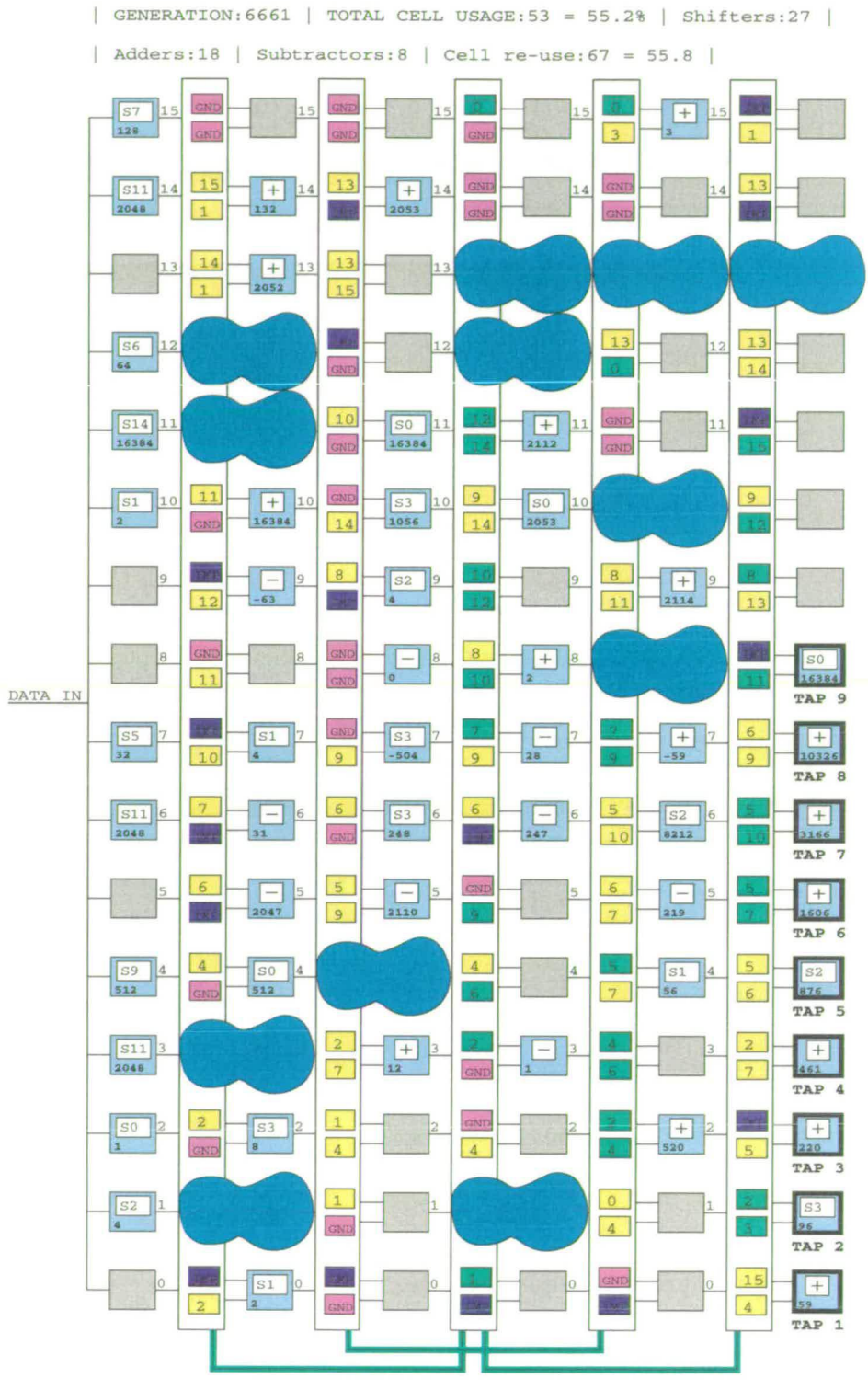


Figure 7.8: *Schematic showing configuration of low-pass FIR filter on PLA with 13% faults.*

displays the results obtained. The averaged evolution of filter fitness using random population initialisation has also been shown.

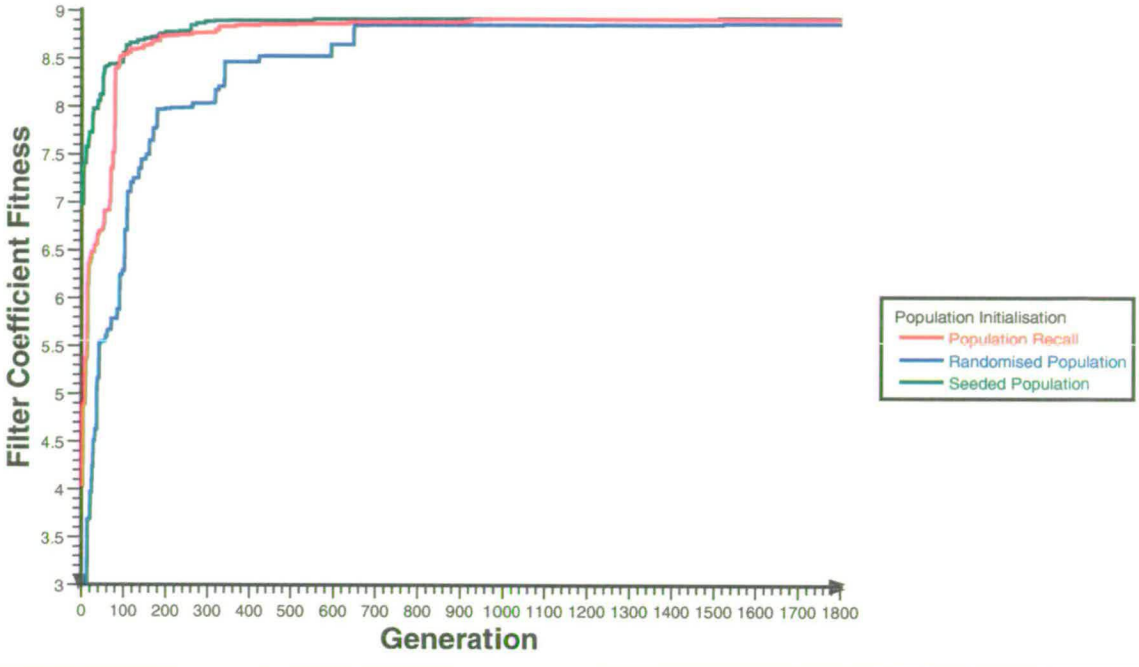


Figure 7.9: *Fitness performance of filter evolved on PLA based on various methods of generating the initial population of configuration-strings.*

Both population seeding and recall enable the GA to adapt to the faulty PLA architecture and produce coefficient sets with target fitness considerably faster than when a population of configuration-strings are randomly generated. Population seeding produces filters of fitness $\geq$ 98.5% after 272 generations, population recall required 513 generations, whilst random initialisation took an average of 1521 generations to reach target fitness. This translates to a 6 fold and 3 fold increase in fault recovery over that of random initialisation for population seeding and recall respectively.

7.4 Summary

The *Col2* PLA architecture, identified as the most effective for the autonomous implementation of FIR filter coefficients, was translated into a synthesisable, technology dependent VHDL netlist. The *ideal interconnect* between columns of PALU was instead replaced with more restricted interconnect reflecting realistic PALU fan-out. The resulting PLA architecture, termed

Col2_reduced, was then compared with *Col2* using the benchmark low-pass filter; with results showing comparable fitness in the coefficients sets produced by the GA.

Six columns and five rows of PALU were selected as the base dimensions to generate a synthesised VHDL core of the *Col2_reduced* PLA architecture. Operational speeds of 10 to 100MHz were presented after synthesis, and reflect typical SoC bus frequencies. A signal latency of 7 clock cycles, independent of filter length, is experienced on the core. This is a considerable advantage over single multiply accumulate DSP architectures which have processing times directly proportional to filter tap length. More complex FIR filters can be constructed by simply adding a number of 6x5 PLA cores during initial VHDL parameterisation of the filter specification.

The ability of the platform to adapt to increasing numbers of faults was investigated through the evolution of a 31-tap low-pass FIR filter. Four increasingly large numbers of faulty PALUs were introduced to the PLA from 0 to 25% of the total architecture. Results show that the functionality of filters evolved on the PLA was maintained despite the increasing number of faults present. This was attributed to redundant PALUs (inherent in the PLA) exploited through the use of EHW. Two additional methods of population initialisation were examined to see if fault recovery times after detection could be decreased compared with random population initialisation. It was shown that seeding a population of random configuration-strings with the best configuration currently obtained produced filters of acceptable fitness 6 times faster than with purely randomised population initialisation.

Chapter 8
Summary and Conclusions

8.1 Introduction

The focus of this thesis has been to investigate if a programmable platform tailored for evolvable hardware (EHW) can be developed which is highly suited to the autonomous implementation of digital FIR filters. Three novel programmable platforms have been developed for this purpose each with a distinctly different architectural design to accommodate FIR coefficient multiplication.

This chapter is organised as follows: Section 8.2 presents a summary of the material presented in each chapter of this thesis, Section 8.3 then provides conclusions drawn from the data collected, supporting or rejecting the thesis statement given above. Section 8.4 highlights what has been achieved as a result of the research carried out and Section 8.5 outlines future work which would add to the knowledge already gained from research undertaken in this thesis, and discuses what might be done to improve the performance of the EHW platforms. Finally section 8.6 presents a number of final comments regarding the thesis, and possible applications of research presented therein.

8.2 Summary

The underlying theme of this thesis has been to investigate a number of programmable architectures for the automated design of digital FIR filters using the principle of EHW. Each programmable architecture is therefore configured using a genetic algorithm (GA) which is derived from a class of non-heuristic search and optimisation techniques termed evolutionary algorithms, which are inspired by the process of biological evolution. The GA must therefore successfully search for and manipulate the encoding used to configure each programmable architecture in order to generate the desired filter coefficient set in hardware.

Chapter 2 introduced the concepts behind evolutionary algorithms and identified four distinct classes, one of which was the GA. The suitability of the GA for automated digital circuit design

using EHW was identified due to the manner in which possible circuit solution are encoded and manipulated. The mechanisms behind the algorithms operation were also presented and focused on the application of autonomous circuit design.

Chapter 2 also demonstrated the benefits and limitations behind both gate-level and functional-level approaches to EHW circuit design through literature review. The differences between software-based circuit simulation (extrinsic evaluation), and hardware-based, or intrinsic circuit evaluation were also evaluated. It was shown that extrinsic evaluation techniques lack models which describe the physical characteristic of the circuit evolved, whilst intrinsic evaluation often resulted in the exploitation of anomalous physical characteristics particular to the device on which the circuit was evolved.

The first of three EHW platforms termed the Virtual Chip was presented in Chapter 3. Circuits were generated in the Virtual Chip using a GA which described each circuit as a VHDL model. Each model was then evaluated extrinsically using simulation tools designed to take account of physical circuit characteristics such as timing and area. The Virtual Chip was initially used to compare the effectiveness of two circuit component libraries; one reflecting gate-level evolution, the other function-level by using a GA to autonomously design three types of DSP circuit: an MxN multiplier, 7-bit one's voter and a 2-tone frequency discriminator, all of which had been previously benchmarked in the EHW community.

The concept of phased evolution is also introduced in Chapter 3 as an approach to generate more complex multiplier circuits for FIR coefficient multiplication. Phased evolution partitions circuit complexity into relevant circuit outputs and in doing so reduces the search space into smaller landscapes which relate to each sub-circuit. A 3x3 bit parallel multiplier was generated using phased evolution that could not be generated in the same number of generations using the conventional Virtual Chip

Chapter 4 presented the basic concepts behind FIR filter theory and demonstrated how filters can be implemented in hardware using the direct-form and transposed-direct-form. The multiplier was identified as the most costly component in filter implementation and a number of design methodologies were discussed which either reduced the role of the multiplier, or replaced it with a series of bit-shifts, additions and subtractions. In particularly, the primitive operator filter (POF) methodology was identified as a technique suitable for adaptation to EHW using functional-level evolution. A wide range of fixed-function and programmable

VLSI architectures dedicated to implementing FIR filters, with and without explicit multiplication, were also presented. In addition, a class of general purpose programmable logic devices (PLDs) were introduced and methods for performing coefficient multiplication on theses architectures investigated.

From the information presented in Chapter 4, Chapter 5 detailed the development of a PALU dedicated to FIR coefficient multiplication using the POF approach. The PALU was designed to be implemented in two distinct array structures, reflecting two different classes of PLD. The GA used to configure each PALU was also presented. Both the GA and PALU were written in VHDL and formed the backbone of the final two programmable EHW platforms.

Chapter 6 presented the development and evaluation of the final two programmable platforms, dedicated to FIR coefficient multiplication. The first was based on a standard FPGA architecture, the second on a conventional PLA. Both PLDs were investigated using a number of filter input, tap output and PALU interconnect topologies. A 31-tap low-pass filter was used to provide a benchmark for comparison between each programmable platform and topology variation. Each PLA and FPGA topology was analysed based on a number of performance criteria such as the quality of filter response produced by the coefficient set, and the number of PALUs utilised in any given array.

The most effective PLA-based filter architecture identified in chapter 6 was translated into a technology dependent netlist in Chapter 7. Physical constraints were examined an modifications to the original architecture made where necessary. The ability of the platform to adapt to increasing levels of faulty PALUs was also investigated. Results show that the quality of coefficients evolved on the netlisted PLA was maintained. Three approaches to population seeding were compared to see which most aided fault recovery times after detection.

8.3 Conclusions

This thesis proposed that a programmable platform tailored for evolvable hardware can be developed which is highly suited to the autonomous implementation of digital FIR filters. From information presented in Chapter 2 it can be concluded that a genetic algorithm provides suitable search mechanisms for developing digital circuits using the EHW approach. Gate-level evolution has been shown to produce digital circuits smaller in area than those developed using conventional design techniques. However gate-level evolution fails on more complex circuits

as the search space which must be successfully navigated to generate them grows non-linearly with circuit complexity. It was shown that functional-level evolution can be used to constrain the search space by using larger circuit building blocks, providing the genetic algorithm with an easier means of finding solutions for more complex digital circuits.

From results presented of the Virtual Chip EHW platform developed in Chapter 3 it can be concluded that functional-level evolution considerably outperforms gate-level by enabling the genetic algorithm to successfully generate more solutions for each of the DSP circuits investigated. In addition, it can be concluded that for all circuits investigated on the Virtual Chip the genetic algorithm required less time to generate circuit solutions using a functional-level component library than when the gate-level component library was employed. Because fewer logic elements are required to encode circuit descriptions using the functional library, this results in a search space several orders of magnitude smaller than that produced by the longer circuit encodings required when using the gate-level library. The timing and area characteristics of the circuits generated by both the functional and gate-level component libraries were comparable. However in both cases the GA produced circuits which were either equal to or better in performance than functionally equivalent circuits generated using standard digital design techniques. It can therefore be concluded that, for the DSP circuits evaluated, functional-level evolution does not result in the generation of circuits with lower performance in terms of physical area than those produced using simple gate primitives.

Phased evolution, also presented in Chapter 3, partitions circuit complexity and in doing so reduces the search space into smaller landscapes, related to each sub-circuit. It can therefore be concluded that this segmented approach reduces the associated degree of epistasis inherent in the chromosomes circuit encoding. This make it possible to evolve complex multiplier circuits more effectively than simply evolving the circuit as a single entity. Results also demonstrate the non-uniformity in the complexity of the multiplier architecture related to individual output paths. It can be concluded that multiplier circuits generated using phased evolution are of equivalent performance in terms of area and timing to multipliers generated using standard design techniques, in addition to other published design techniques using EHW. However failure of the Virtual Chip to generate a 4x4 bit parallel multiplier when using phased evolution indicates that larger logic components of greater functionality are required to generate a multiplication unit of sufficient complexity to implement FIR coefficient multiplication. In addition, the success of other published EHW approaches indicates that a more constrained programmable architecture

is required to further reduce the multiplier search space.

Chapter 6 presented the development of two programmable platforms inspired by two different classes of PLD. Each platform was specifically designed to implement FIR coefficient multiplication using a distributed, multiplierless architecture based around primitive operator filters (POF). Initial results support the conclusion that crossover was not effective in enabling the genetic algorithm to configure either programmable platform to implement the specified coefficient set. This is due to the high level of epistasis inherent in the POF design problem. This high degree of epistasis means that interactions between PALUs and programmable interconnects is non-linear, and filter fitness cannot be directly attributed to the effects of an individual PALU.

From a number of PLA and FPGA-based filter topologies examined, it can be concluded that the PLA-based filter architecture considerably outperformed the FPGA in terms of the quality of coefficients sets produced. This is supported by results in Chapter 6 which show that on average the PLA-based architectures produced coefficients with a fitness score 4% higher than those produced on the FPGA. It can further be concluded that the performance of the PLA is attributed to the higher degree of flexibility afforded by the PLA interconnect topologies which utilise hierarchical connectivity, and that the critical path of the PLA is in every case shorter than the FPGA. These two factors have been shown to reduce linkage between PALUs, indicated by reductions in PALU re-use, which improves performance. It has therefore been shown that a PLA architecture implementing column-based tap placement with a nearest neighbour and 2-level PALU interconnect hierarchy is the most effect programmable platform for the autonomous implementation of FIR filter coefficient multiplication using EHW.

From translating the most effective PLA architecture identified into a synthesisable, physically realisable component netlist with reduced PALU interconnect, it can be concluded that no reduction in performance was experience when compared to the original PLA model. The creation of a 6x16 PLA revealed that changing the dimensions of the PLA significantly reduces the latency of the filter to 7 clock cycles, irrespective of tap length, and incurs no reduction in the quality of filter coefficients produced. In fact around 20% fewer PALUs were required to implemented the 31-tap filter benchmarked when the 6x16 PLA was employed. Similar ratios were experienced with PALU re-uses, supporting the conclusion that lower PALU linkage results in better PLD performance.

The fault tolerance of the 6x16 PLA was investigated in Chapter 7. Results show that the quality of filter coefficients evolved on the PLA can be maintained despite a 25% increase in the number of faulty PALUs present on the array. It can therefore be concluded that the 6x16 PLA provides sufficient PALU redundancy to enable the GA to over come the inclusion of a significant number of faults. Three approaches to population seeding were compared: random initialisation, population seeding and population recall. Each approach was examined to see which most aided fault recovery times after faults were detected on the PLA. Results show that population seeding reproduced coefficients that were of acceptable fitness 6 times faster than when random initialisation was used and 3 time faster than population recall. It can then be concluded that population seeding provides the most effective means of adapting a population of configuration strings to over come faults on the 6x16 PLA for a given set of filter coefficients.

8.4 Achievements

The research presented in this thesis has required the development of a number of software and hardware-based models and programs. For completeness these are highlighted below:

- The development of a novel genetic algorithm written in C for the Virtual Chip EHW platform.
- The creation of the Virtual Chip EHW environment used to autonomously generate VHDL descriptions of *evolving* circuit solutions.
- The development of a Programmable Arithmetic Logic Unit (PALU) written and parameterised in VHDL and inspired by the concept of the primitive operator filter (POF) approach to implementing FIR filters.
- The development of an FPGA and PLA-inspired array of PALUs dedicated to programmable FIR filter coefficient multiplication. The programmable arrays were written in VHDL and designed to express a number of interconnects and filter input and output topologies which could then be configured using a GA.
- Generation of a parametrisable genetic algorithm, written in VHDL and embedded alongside the FPGA and PLA-based programmable platforms.

- Development of visualisation software written in C and designed to model both the FPGA and PLA-based architectures and graphically depict the filter configurations produced. This was achieved using postscript format.

8.5 Future Work

This thesis has endeavoured to provide a rigorous investigation of the focus of research outlined in section 8.1. However, a number of additional potentially interesting areas remain which might further add to the knowledge already gained from the research presented.

The suitability of each coefficient string used to configure the two PLD-based programmable platforms is currently calculated by the fitness of the coefficient set produced. Filter performance might further be improved by including a Pareto-based fitness measure incorporating all four performance criteria: PALU utilisation, PALU re-use, the ratio of additions, subtractions and bit-shifts; as well as coefficient fitness. Whilst this would not directly add to a more efficient comparison between EHW platforms, it might provide more optimised filter implementations.

Another method of improving the effectiveness in which the GA can configure each programmable platform would be to add heuristic knowledge of both the programmable architecture and the POF design approach into the GA search function. This might for example utilise the directed graph/GA hybrid approach implemented by Redmill et.al in [97].

An investigation as to the suitability of other search techniques such as simulated annealing for autonomously implementing filter coefficients on each of the three programmable platform might also be performed. This would provide a useful evaluation of the effectiveness of EHW for the automated digital filter design problem presented in this thesis.

Finally, it is possible to extend the Virtual EHW platform to enable POF-based coefficient multiplication. This would then provide a means of accurately appraising the performance of the Virtual Chip when compared to the PLA and FPGA-based EHW platforms.

8.6 Final Comments

In summary, this thesis has investigated whether a programmable platform tailored for evolvable hardware can be developed which is highly suited to the autonomous implementation of digital

FIR filters. It can therefore be concluded that the 6xN PLA architecture developed in Chapter 7 provides the most suitable platform for evolving coefficient taps in terms of the quality of filter coefficient produced, the PALU resources utilised, the overall latency of the filter implemented, and the architectures resilience to faults through controlled redundancy.

High performance digital filters are in great demand throughout the communication industry and other sectors that require data control and manipulation. Industrial requirements include fast operational speed, low physical area, device portability, and reliability/robustness. The PLA-based EHW platform developed satisfies many of these conditions, and could be embedded into a number of SoC devices that might benefit from online adaptive data manipulation.

References

[1] Z. Yajlang, H. Lingyi, Q. Yulin, X. Xia, and H. Xiaoling, "A high speed multiplication-and-accumulation design methodology for submicron and deep submicron dsp solutions," in *5th IEEE Int. Conf. on Solid-State and Integrated Circuit Technology*, pp. 502–504, 1998.

[2] D. R. Bull and D. H. Horrocks, "Primitive operator digital filters," in *IEE Proc -G*, pp. 401–412, 1991.

[3] V. M. Porto, "Evolutionary methods for training neural networks for underwater pattern classification," in *24th Ann. Asilomar Conf. on Signals, Systems and Computers*, vol. 2, pp. 1015–1019, 1989.

[4] D. B. Fogel, L. J. Fogel, and V. M. Porto, "Evolving neural networks," *Biol. Cybernet.*, vol. 63, pp. 487–93, 1990.

[5] P. Angeline, G. Saunders, and J. Pollack, "Complete induction of recurrent neural networks," in *Proc. 3rd Ann. Conf. on Evolutionary Programming*, pp. 1–8, 1994.

[6] K. P. Dahal, G. M. Burt, J. R. NcDonald, and A. Moyes, "A case study of scheduling storage tanks using a hybrid genetic algorithm," *IEEE Transactions on Evolutionary Computation*, vol. 5, pp. 283–294, June 2001.

[7] R. Chandrasekharam, S. Subhramanian, and S. Chaudhury, "Genetic algorithm for node partitioning problem and applications in vlsi design," in *IEE Proceedings Computers and Digital Techniques*, vol. 140, pp. 255–260, Sept 1993.

[8] B. M. Goni and T. Arslan, "An evolutionary 3d over-the-cell router," in *12th Annual IEEE International ASIC/SOC Conference*, (Washington, DC.), pp. 206–209, Sept 15-18 1999.

[9] J. Arabas and S. Kozdrowski, "Applying an evolutionary algorithm to telecommunication network design," *IEEE Trans. on Evolutionary Computation*, vol. 5, pp. 309–322, Aug 2001.

[10] R. Drechsler, *Evolutionary Algorithms for VLSI CAD*. Boston MA: Kluwer, ISBN 0-7923-8168-8, 1998.

[11] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, and K. Toda, "Real-world applications of analog and digital evolvable hardware," *IEEE Transactions on Evolutionary Computation*, vol. 3, pp. 220–235, September 1999.

[12] V. K. Vassilev, D. Job, and J. F. Miller, "Towards the automatic design of more efficient digital circuits," in *In Proceedings of the Second NASA/DOD Workshop on Evolvable Hardware*, pp. 151–160, 2000.

[13] A. Thompson, "An evolved circuit, intrinsic in silicon entwined with physics," in *Evolvable Systems: From Biology to Hardware. (ICES 96)*, pp. 390–405, 1996.

[14] X. Yao and T. Higuchi, "Promises and challenges of evolvable hardware," in *Evolvable Systems: From Biology to Hardware. (ICES 96)*, pp. 55–80, 1996.

[15] M. Sipper and D. Mange, "Guest editorial from biology to hardware and back," *IEEE Transactions on Evolutionary Computation*, 1999.

[16] K. Imamura, J. A. Foster, and A. W. Krings, "The test vector problem and limitations to evolving digital circuits," in *In Proceedings of the Second NASA/DOD Workshop on Evolvable Hardware*, pp. 75–79, 2000.

[17] G. Tufte and P. C. Haddow, "Evolving and adaptive filter," in *In Proceedings of the Second NASA/DOD Workshop on Evolvable Hardware*, pp. 143–150, 2000.

[18] S. Levi and A. K. Agrawala, *Fault tolerant system design.* McGraw-Hill, 1994.

[19] A. Thompson and P. Layzell, "Analysis of unconventional evolved electronics," *Communications of the ACM*, vol. 42, pp. 71–79, Apr. 1999.

[20] A. M. Tyrell, G. Hollingworth, and S. L. Smith, "Evolutionary strategies and intrinsic fault tolerance," in *In Proceedings of the Third NASA/DOD Workshop on Evolvable Hardware*, pp. 98–108, July 2001.

[21] T.-S. Park, C.-H. Lee, and D.-J. Chung, "Intrinsic evolution for synthesis of fault recoverable circuit," *IEICE Trans. Fundamentals*, pp. 2488–2497, Dec. 2000.

[22] A. Stoica, D. Keymeulen, and R. Zebulum, "Evolvable hardware solutions for extreme temperature electronics," in *In Proceedings of the Third NASA/DOD Workshop on Evolvable Hardware*, pp. 93–97, July 2001.

[23] M. Salami, H. Sakanashi, M. Tanaka, M. Iwata, T. Kurita, and T. Higuchi, "On-line compression of high precision printer images by evolvable hardware," in *Proceedings of Data Compression Conference. DCC '98*, pp. 219–228, 1998.

[24] www.xilinx.com, "Xilinx data book 2000."

[25] A. T. G. Fuller and B. Nowrouzian, "A novel technique for optimization over the canonical signed-digit number space using genetic algorithms," in *IEEE Int. Sypm. Circuits and Systems, ISCAS*, vol. 2, pp. 745–748, 2001.

[26] G. Wade, A. Roberts, and G. Williams, "Multiplier-less fir filter design using a genetic algorithm," *IEE Proc. Vision, Image and Signal Processing*, vol. 141, pp. 175–180, June 1994.

[27] B. I. Hounsell and T. Arslan, "A novel evolvable hardware framework for the evolution of high performance digital circuits," in *In Proceedings of GECCO 2000*, vol. 1, (Las Vegas USA), pp. 525–532, July 2000.

[28] B. I. Hounsell and T. Arslan, "A novel genetic algorithm for the automated design of performance driven digital circuits," in *In Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, vol. 1, (La Hoya USA), pp. 601–608, July 2000.

[29] B. I. Hounsell and T. Arslan, "Evolutionary design and adaptation of digital filters within an embedded fault tolerant hardware platform," in *Proceedings of 3rd NASA/DoD IEEE workshop on Evolvable Hardware*, vol. 1, (Los Angeles USA), pp. 127–135, July 2001.

[30] B. I. Hounsell and T. Arslan, "A programmable multiplierless digital filter array for embedded soc application," *In IEE Electronics Letters*, vol. 37, pp. 737–737, June 2001.

[31] B. I. Hounsell and T. Arslan, "An embedded programmable core for the implementation off high performance digital filters," in *In Proceedings of 14th Annual IEEE International ASIC/SoC Conference,*, (Washington USA), pp. 12–14, Sept 2001.

[32] B. I. Hounsell and T. Arslan, "n embedded programmable logic array for online adaptation of multiplierless fir filters," *Submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2001.

[33] E. F. Moore, "Gedanken-experiments on sequential machines: automata studies," in *Annals of Mathematical Studies*, vol. 34, pp. 129–153, Princeton, NJ: Princeton University Press, 1957.

[34] G. H. Mealy, "A method of synthesizing sequential circuits," *Bell Syst. Tech. J.*, vol. 34, pp. 1054–79, 1955.

[35] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial intelligence through simulated evolution*. New York: Wiley, 1966.

[36] D. B. Fogel and J. L. Fogel, "Optimal routing of multiple autonomous underwater vehicles through evolutionary programming," in *IEEE Proc. Symp. on Autonomous Underwater Vehicle Technology*, pp. 44–47, 1990.

[37] W. C. Page, J. M. McDonnell, and B. Anderson, "An evolutionary programming approach to multi-dimensional path planning," in *Proc. 1st Ann. Conf. on Evolutionary Programming*, pp. 63–70, 1992.

[38] P. G. Harrald and D. B. Fogel, "Evolving continuous behaviours in the iterated prisoner's dilemma," *BioSystems*, vol. 37, pp. 135–145, 1996.

[39] D. B. Fogel, "The evolution of intelligent decision making in gaming," *Cybernet. Syst.*, vol. 22, pp. 223–236, 1991.

[40] D. B. Fogel, "Applying evolutionary programming to selected travelling salesman problems," *Cybernet. Syst.*, vol. 24, pp. 27–36, 1993.

[41] I. Rechenberg, "Evolutionsstrategien," in *Simulationsmethoden in der Medizin und Biologie* (B. Schneider and U. Ranft, eds.), pp. 83–114, Berlin: Springer, 1978.

[42] T. Bäck, *Evolutionary Algorithms in theory and Practice. Evolutionary Strategies Evolutionary Programming Genetic Algorithms*. Oxford University Press, 1996.

[43] J. R. Koza, "Hierarchical genetic algorithms operating on populations of computer programs," in *Proc. 11th Int. Joint Conf. on Artificial Intelligence*, San Mateo, CA: Morgan Kaufmann, 1989.

[44] N. L. Cramer, "A representation of the adaptive generation of simple sequential programs," in *Proc. 1st Int. Conf. on Genetic Algorithms* (J. J. Grefenstette, ed.), Hillsdale, NJ: Erlbaum, July 1985.

[45] J. R. Koza, F. H. B. I. amd D. Andre, M. A. Keane, and F. Dunlap, "Automated synthesis of analog electrical circuits by means of genetic programming," *EEE Trans. on Evolutionary Computation*, vol. 9, pp. 109–128, July 1997.

[46] D. Andre, F. H. B. III, J. Koza, and M. A. Keane, "On the theory of designing circuits using genetic programming and a minimum of domain knowledge," in *EEE World Congress on Computational Intelligence.*, pp. 130–135, 1998.

[47] M. Brameier and W. Banzhaf, "A comparison of linear genetic programming and neural networks in medical data mining," *IEEE Trans. on Evolutionary Computation*, vol. 5, pp. 17–26, Febuary 2001.

[48] P. S. Negan, M. L. Wong, K. S. Leung, and J. C. Y. Cheug, "Using grammar based genetic programming for data mining of medical knowledge," in *Proc. 3rd Annu. Conf. on Genetic Pogramming*, 1998.

[49] R. B. Nachbar, "Molecular evolution: Automated manipulation of hierarchical chemical topology and its application to average molecular structures," *Genetic Programming and Evolvable Machines*, vol. 1, pp. 57–94, April 2000.

[50] J. Koza, *Genetic Programming: On the programming of Computers by means of natural selection*. MIT Press, 1992.

[51] K. E. K. Jr, ed., *Advances in Genetic Programming*. Cambridge MA: MIT Press, 1994.

[52] J. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[53] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[54] T. Bäck, D. B. Fogel, and T. Michalewicz, eds., *Evolutionary Computation 1 Basic Algorithms and Operations*. Institute of Physics Publishing, ISBN 0750306645, 2000.

[55] P. Thompson, "Circuit evolution and visualisation," in *Evolvable Systems: From Biology to Hardware. (ICES 2000)*, pp. 229–240, 2000.

[56] M. Pedram, "Power minimisation in ic design: Principles and applications," *ACM Transations on Design Automation of Electronic Systems*, vol. 1, no. 1, pp. 3–56, January 1996.

[57] V. K. Vassilev and J. F. Miller, "Scalability problems of digital circuit evolution," in *In Proceedings of the Second NASA/DOD Workshop on Evolvable Hardware*, pp. 55–64, 2000.

[58] V. K. Vassilev, J. F. Miller, and T. C. Fogarty, "On the nature of two-bit multiplier landscapes," in *In Proceedings of the First NASA/DOD Workshop on Evolvable Hardware*, pp. 36–45, 1999.

[59] M. Murakawa, "Hardware evolution at functional level," in *Proc. of the international Conference on Parallel Problem Solving from Nature (PPSN'96)*, pp. 62–71, 1996.

[60] I. Kajitani, T. Hoshino, D. Nishikawa, H. Yokoi, S. Nakaya, T. Yamauchi, T. Inuo, N. Kajihara, M. Iwata, D. Keymeulen, and T. Higuchi, "A gate-level ehw chip: Implementing ga operations and reconfigurable hardware on a single sli," in *Evolvable Systems: From Biology to Hardware. (ICES 98)*, pp. 1–12, 1998.

[61] E. Ozdemir, *Evolutionary methods for the design of digital circuits and systems.* PhD thesis, The University of Wales, Cardiff, 1999.

[62] M. Tanaka, H. Sakanashi, M. Salami, M. Iwata, T. Kurita, and T. Higuchi, "Data compression for digital color electrophotographic printer with evolvable hardware," in *Evolvable Systems: From Biology to Hardware. (ICES 98)*, 1998.

[63] M. Murakawa, S. Yoshizawa, and T. Higuchi, "Adaptive equalization of digital communication channels using evolvable hardware," in *Evolvable Systems: From Biology to Hardware. (ICES 96)*, pp. 379–389, October 1996. Functional level approach to EHW using taylored GA/hardware chip.

[64] R. S. Zebulum, M. A. Pacheco, and M. Vellasco, "Evolvable systems in hardware design taxonomy, survey and applications," in *Evolvable Systems: From Biology to Hardware. (ICES 96)*, pp. 344–358, 1996.

[65] A. Hernandez-Aguirre, C. A. Coello, and B. P. Buckles, "A genetic programming approach to logic function synthesis by means of multiplexers," in *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pp. 46–53, 1999.

[66] R. Drechsler and W. G. unther, "Evolutionary synthesis of multiplexor circuits under hardware constraints," in *In Proceedings of Genetic and Evolutionary Computation Conference (GECCO-2000*, pp. 513–518, 2000.

[67] T. Arslan, D. H. Horrocks, and E. Ozdemir, "Structural cell-based vlsi circuit design using a genetic algorithm," in *IEEE International Symposium on Circuits and Systems*, (Atlanta, USA), pp. 308–311, 1996.

[68] J. F. Miller and P. Thompson, "Aspects of digital evolution: Geometry and learning," in *Evolvable Systems: From Biology to Hardware. (ICES 98)*, pp. 25–35, 1998.

[69] A. Thompson, "On the automatic design of robust electronics through artificial evolution," in *Evolvable Systems: From Biology to Hardware. (ICES 98)*, pp. 13–24, 1998.

[70] P. Layzell, "Reducing hardware evolution's dependency on fpgas," in *Proceedings of the Seventh International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems. MicroNeuro '99*, pp. 171–178, 1999.

[71] D. Levi and S. A. Guccione, "Geneticfpga: Evolving stable circuits on mainstream fpgas," in *In Proceedings of the First NASA/DOD Workshop on Evolvable Hardware* (A. Stoica, D. Keymeulen, and J. L. (Eds.), eds.), pp. 12–17, IEEE Computer Society Press, Los Alamitos, July 1999.

[72] G. Tufte and P. C. Haddow, "Prototyping a ga pipeline for complete hardware evolution," in *Proceedings of The First NASA/DoD Workshop on Evolvable Hardware*, pp. 18 –25, 1999.

[73] A. Hamilton, K. Papathanasiou, M. Tamplin, and T. Brandtner, "Palmo: Field programmable analogue and mixed-signal vlsi for evolvable hardware," in *Evolvable Systems: From Biology to Hardware. (ICES 98)*, pp. 335–344, 1998.

[74] A. Hamilton, P. Thompson, and M. Tamplin, "Experiments in evolvable filter design using pulse based programmable analogue vlsi models," in *Evolvable Systems: From Biology to Hardware. (ICES 2000)*, pp. 61–71, 2000.

[75] A. Stoica, R. Zebulum, D. Keymeulen, R. Tawel, T. Daud, and A. Thankoor, "Reconfigurable vlsi architectures for evolvable hardware: From experimental field programmable transistor arrays to evolution-oriented chips," *IEEE Trans on Very Large Scale Integration (VLSI) Systems*, vol. 9, pp. 227–232, February 2001.

[76] P. J. Ashenden, *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, Inc, 1995.

[77] T. Hikage, H. Hemmi, and K. Shimohara, "Hardware evolution system introducing dominant and recessive heredity," in *Evolvable Systems: From Biology to Hardware. (ICES 96)*, pp. 423–436, Springer, October 1996.

[78] A. Thompson, P. Layzell, and R. S. Zebulum, "Explorations in design space: Unconventional electronics design through artificial evolution," *IEEE Transactions on Evolutionary Computation*, vol. 3, pp. 267–196, September 1999.

[79] Cadence Design Systems, Inc., *BuildGates User Guide*, release 2.3 ed., March 1999.

[80] J. Bergeron, *Writing Testbenches Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000.

[81] V. K. Vassilev, J. F. Miller, and T. C. Fogarty, "Digital circuit evolution and fitness landscapes," in *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 99*, vol. 2, pp. 1299–1306, 1999.

[82] Y. Davidor, "Epistasis variance - suitability of a representation to genetic algorithms," tech. rep., The Weizmann Institute of Science, Dept of Applied Mathematics and Computer Science, December 1989.

[83] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing*, ch. 7, pp. 470–485. Macmillan Publishing Company, NY, 2 ed., 1992.

[84] B. Mulgrew, P. Grant, and J. Thompson, *Digital Signal Processing Concepts and Applications*. MacMillan Press LTD, 1999.

[85] N. M. Mitrou, "Results on nonrecursive digital filters with nonequidistant taps," *IEEE Trans. Acoust., Speach, Signal Processing*, vol. 33, pp. 1621–1624, Dec. 1985.

[86] R. J. Hartnett, "Design of efficient parallel hybrid fir filters using dynamic programming and subset selection methods," in *in Proc. 1990 Int. Conf. Acoust., Speach, Signal Processing*, pp. 1337–1340, 1990.

[87] J. T. Kim, W. J. Oh, and Y. H. Lee, "Design of nonuniformly spaced linear-phase fir filters using mixed integer linear programming," *IEEE Trans. Signal Processing*, vol. 44, pp. 123–126, Jan. 1996.

[88] A. Avizienis, "Signed-digit number representation for fast parallel arithmetic," *IRE Trans. Electon. Comput*, pp. 389–400, 1961.

[89] H. Samueli, "An improved search algorithm for the design of multiplierless fir filters with powers-of-two coefficients," *IEEE Trans. Circuits Syst*, vol. 36, pp. 1044–1047, July 1989.

[90] X. Xu and B. Nowrouzian, "Local search algorithm for the design of multiplierless digital filters with csd multiplier coefficients," in *IEEE Canadian Conference on Electrical and Computer Engineering*, vol. 2, pp. 811–816, May 1999.

[91] Y. C. Lim and S. R. Parker, "Fir filter design over a discrete powers-of-two coefficient space," *IEEE Trans. Acoust., Speach, Signal Processing*, vol. 31, pp. 583–591, June 1983.

[92] Q. Zhao and Y. Tadokoro, "A simple desing of fir filters with powers-of-two coefficients," *IEEE Trans. Circuits Syst*, vol. 35, pp. 566–570, May 1988.

[93] P. Gentili, F. Piazza, and A. Uncini, "Efficient genetic algorithm design for power-of-two fir filters," in *Int Conf. Acoustics, Speech, and Signal Processing. ICASSP-95*, vol. 2, pp. 1268–1271, 1995.

[94] S. Sriranganathan, D. R. Bull, and D. W. Redmill, "Design of 2-d multiplierless fir filters using genetic algorithms," in *First Int Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications. GALESIA*, pp. 282–286, 1995.

[95] G. Wacey and D. R. Bull, "Architectural synthesis of digital filters for asic implementation," in *Digital and Analogue Filter and Filtering Systems, IEE Colloquium on*, pp. 6/1–6/5, 1991.

[96] T. Arslan, H. I. Eskikurt, and D. H. Horrocks, "Configurable structures for a primitive operator digital filter fpga," in *IEEE Workshop on Signal Processing Systems, SIPS 97 - Design and Implementation*, pp. 532–540, 1997.

[97] D. W. Redmill, D. R. Bull, and E. Dagless, "Genetic synthesis of reduced complexity filters and filter banks using primitive operator directed graphs," *IEE Proc. -Circuits Devices Syst*, vol. 147, pp. 303–310, Oct. 2000.

[98] D. R. Bull, *Reduced complexity*. PhD thesis, School of Electronic and Systems Engineering, University of Wales, 1989.

[99] R. A. Hawley, B. C. Wong, T. ji Lin, J. Laskowski, and H. Samueli, "Design techniques for silicon compiler implementations of high-speed fir digital filters," *IEEE Journal. Sold-State Circuits*, vol. 31, pp. 656–667, May 1996.

[100] J. B. Evans, "An efficient fir filter architecture," in *in Proc. Int. Symposium. Acoust., Speach, Signal Processing*, vol. 1, pp. 627–630, May 1993.

[101] I. E. Ungan and M. Askar, "A gate array chip for high frequency dsp applications," in *in Proc. Int. Conf. 7th Mediterranean Electrotechnical*, pp. 549–552, 1994.

[102] S. Nooshabadi, J. A. Montiel-Nelson, and G. S. Visweswarain, "Micropipeline architecture for multiplier-less fir filters," in *in Proc. Int. Conf. 10th Conference on VLSI Design*, pp. 451–456, Jan. 1997.

[103] S. Yoon and M. H. Sunwoo, "An efficient multiplierless fir filter chip with variable-length taps," in *IEEE Workshop on Signal Processing Systems. SIPS 97 - Design and Implementation*, pp. 412–420, 1997.

[104] K.-Y. Khoo, A. Kwentus, and J. A. N. Wilson, "An efficient 175mhz programmable fir digital filter," in *IEEE Int Symp on Circuits and Systems, ISCAS '93*, pp. 72–75, 1993.

[105] K.-Y. Khoo, A. Kwentus, and J. A. N. Wilson, "A programmable fir digital filter using csd coefficients," *IEEE Journal. Sold-State Circuits*, vol. 31, pp. 869–874, June 1996.

[106] W. J. Oh and Y. H. Lee, "Implementation of programmable multiplierless fir filters with powers-of-two coefficients," *IEEE Trans. Circuits Syst*, vol. 42, pp. 553–556, Aug. 1995.

[107] S. R. Powell and P. M. Chau, "Reduced complexity programmable fir filters," in *IEEE Int Symp on Circuits and Systems. ISCAS '92. Proceedings*, pp. 561–564, 1992.

[108] J. F. Miller, "On the filtering properties of evolved gate arrays," in *In Proceedings of the First NASA/DOD Workshop on Evolvable Hardware*, pp. 2–11, 1999.

[109] S. J. Flockton and K. Sheeham, "Behaviour of a building block for intrinsic evolution of analogue signal shaping and filtering circuits," in *In Proceedings of the Second NASA/DOD Workshop on Evolvable Hardware*, pp. 117–123, 2000.

[110] Y. Tamir and M. Tremblay, "High performance fault-tolerant vlsi systems using micro rollback," *IEEE Trans. Computers*, vol. 39, pp. 548–554, Apr. 1990.

[111] J. H. Patel and L. Y. Fung, "Concurrent error detection in alu's by recomputing with shifted operands," *IEEE Trans. Computers*, vol. 31, pp. 589–595, July 1982.

[112] R. Karri, K. Hogstedt, and A. Orailoglu, "Rapid prototyping of fault tolerant vlsi systems," in *Int Symp on High-Level Synthesis. 7th Proc*, pp. 126–131, 1994.

[113] L. Mintzer, "Digital filtering in fpgas," in *Twenty-Eighth Asilomar Conf. on Signals, Systems and Computers*, vol. 2, pp. 1373–1377, 1994.

[114] S. A. White, "Applications of distributed arithmetic to digital signal processing: A tutorial review," *IEEE ASSP Magazine*, vol. 6, no. 3, pp. 4–19, 1989.

[115] M. Martinez-Peiro, J. Valls, T. Sansaloni, A. P. Pascual, and E. I. Boemo, "A comparison between lattice, cascade and direct form fir filter structures by using a fpga bit-serial distributed arithmetic implementation," in *6th IEEE Proc. Int. Conf. Electronics, Circuits and Systems (ICECS'99)*, vol. 1, pp. 241–244, 1991.

[116] V. Pasham, A. Miller, and K. Chapman, "Application notes from virtex and virtex-ii series: Transposed form fir filters," tech. rep., Xilinx, January 10th 2001.

[117] A. Amira, *A custom coprocessor for matrix algorithms*. PhD thesis, University of Belfast, 2001.

[118] D. C. Chen and J. M. Rabaey, "A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths," *IEEE journal of Solid-State Circuits*, vol. 27, pp. 1895–1904, Dec 1992.

[119] K. Rajagopalan and P. Sutton, "A flexible multiplication unit for an fpga logic block," in *Int. Symp on Circuits and Systems (ISCAS) 2001*, vol. 4, pp. 546–549, 2001.

[120] N. Venkateswaran, A. K. Murugavel, and G. Chandramouli, "Field programmable dsp transform arrays," in *IEEE Workshop on Signal Processing Systems (SIPS)*, pp. 152–161, 1998.

[121] R. Porter, k. McCabe, and N. Bergmann, "An applications approach to evolvable hardware," in *In Proceedings of the First NASA/DOD Workshop on Evolvable Hardware*, pp. 170–174, 1999.

[122] M. Sipper, M. Goeke, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini, "The firefly machine: online evolware," in *IEEE Int Conf on Evolutionary Computation*, pp. 181–186, 1997.

[123] Y.-H. Choi and D. J. Chung, "Vlsi procsesor of parallel genetic algorithm," in *Proceedings of the Second IEEE Asia Pacific Conference on ASICs. AP-ASIC 2000*, pp. 143–146, 2000.

[124] T. Higuchi, M. Masahiro, M. Iwata, I. Kajitani, W. Liu, and M. Salami, "Evolvable hardware at functional level," in *IEEE International Conference on Evolutionary Computation*, pp. 187–192, 1997.

[125] S. Wakabayashi, T. Koide, N. Toshine, M. Goto, Y. Nakayama, and K. Hayya, "An lsi implementation of an adaptive genetic algorithm with on-the-fly crossover operator selection," in *Proceedings of the ASP-DAC '99. Asia and South PacificDesign Automation Conference*, vol. 1, pp. 37–40, 1999.

[126] E. Zwyssig, "Low power digital filter design for hearing aid applications," Master's thesis, The University of Edinburgh UK, 2000.

[127] C. R. Reeves, "Predictive measures for problem difficulty," in *Congress on Evolutionary Computation, CEC*, vol. 1, pp. 736–743, 1999.

[128] P. Merz and B. Freisleben, "On the effectiveness of evolutionary search in high-dimensional nk-landscapes," in *IEEE Int. Conf. on Computational Intelligence. Evolutionary Computation Proceeding*, pp. 741–745, 1998.

[129] H. Tsutsui, K. Hiwada, T. Izumi, T. Onoye, and Y. Nakamura, "A design of lut-array-based pld and a synthesis approach based on sum of generalized complex terms expression," in *IEEE Int. Symp on Circuits and Systems, ISCAS'2001*, vol. 5, pp. 203–206, 2001.

[130] J. McClellan, T. W. Parks, and L. R. Rabiner, "A computer program for designing optimum fir linear phase digital filters," *Transactions on Audio and Electroacoustics*, pp. 506–526, Dec. 1973.

[131] A. Thompson, "Evolving fault tolerant systems," in *First Int. Conf on Genetic Algorithms in Engineering Systems: Innovations and applications. GALESIA*, pp. 524–529, 1995.

[132] A. Stoica, D. Keymeulen, V. Duong, and C. Salazar-Lazaro, "Automatic synthesis and fault-tolerant experiments on an evolvable hardware platform," in *IEEE Aerospace Conference Proceedings*, vol. 5, pp. 465–471, 2000.

[133] J. D. Lohn, G. L. Haith, S. P. Colombano, and D. Stassinopoulos, "Towards evolving circuits for autonomous space applications," in *IEEE Aerospace Conference Proceedings*, vol. 5, pp. 473–486, 2000.

Appendix A
VHDL Code for DSP Circuits

A.1 VHDL gate-level description of 2-bit multiplier

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE WORK.library_cells.ALL;

ENTITY multi_2bit IS
  PORT(SIGNAL in0, in1, in2, in3 : IN std_ulogic;
       SIGNAL out0, out1, out2, out3 : OUT std_ulogic);
END multi_2bit;

ARCHITECTURE struc OF multi_2bit IS
 SIGNAL inter0, inter1, inter2, inter3 : std_ulogic;
 SIGNAL zero : std_ulogic := '0';
BEGIN
  cell_0:AND_2input
    PORT MAP (in0, in2, out0);
  cell_1:AND_2input
    PORT MAP (in1, in2, inter0);
  cell_2:AND_2input
    PORT MAP (in1, in3, inter2);
  cell_3:AND_2input
    PORT MAP (in0, in3, inter1);
  cell_4:FULLADDER
    PORT MAP (inter1, zero, inter0, out1, inter3);
  cell_5:FULLADDER
    PORT MAP (zero, inter3, inter2, out2, out3);
END struc;
```

A.2 7-bit pattern recognizer (one's voter)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.library_cells.ALL;

ENTITY recog_7bit IS
  PORT(SIGNAL in0, in1, in2, in3, in4, in5, in6 : IN std_ulogic;
       SIGNAL out0 : OUT std_ulogic);
END recog_7bit;

ARCHITECTURE struc OF recog_7bit IS
```

```
  SIGNAL inter1, inter2 : std_ulogic;
BEGIN
  cell_0: recog_3bit
    PORT MAP (in0, in1, in2, inter1);
  cell_1: recog_3bit
    PORT MAP (in3, in4, in5, inter2);
  cell_2: recog_3bit
    PORT MAP (inter1, inter2, in6, out0);
END struc;
```

A.2.1 3-bit pattern recognizer

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;

ENTITY recog_3bit IS
  PORT(SIGNAL in0, in1, in2 : IN std_ulogic;
       SIGNAL out0 : OUT std_ulogic);
END recog_3bit;

ARCHITECTURE struc OF recog_3bit IS
BEGIN
    out0 <= ((in0 AND in1) OR (in0 AND in2) OR (in1 AND in2));

END struc;
```

A.3 A behavioural model of a two tonne discriminator

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY bhv_tonne IS
   PORT(SIGNAL freq_in   : IN std_logic;
        SIGNAL clock     : IN std_logic;
        SIGNAL decision  : OUT std_logic);
END bhv_tonne;

ARCHITECTURE bhv OF bhv_tonne IS
BEGIN
  MainBody: PROCESS(clock, freq_in)
    VARIABLE pos_count : integer RANGE 0 TO 255 := 0;
    VARIABLE count_decision : std_logic;
  BEGIN
    IF freq_in = '1'
    THEN
      IF rising_edge(clock)
      THEN
        pos_count := pos_count + 1;
      END if;
    ELSIF freq_in = '0'
```

```
    THEN
      CASE pos_count IS
        WHEN 4 =>
          count_decision := '0';
        WHEN 12 =>
          count_decision := '1';
        WHEN OTHERS =>
          count_decision := count_decision;
      END CASE;
      pos_count := 0;
    END IF;
    decision <= count_decision;
  END PROCESS MainBody;
END bhv;
```

A.4 Schematic of 2x2-bit Parallel Multiplier Evolved by Miller et.al. and Associated VHDL Code

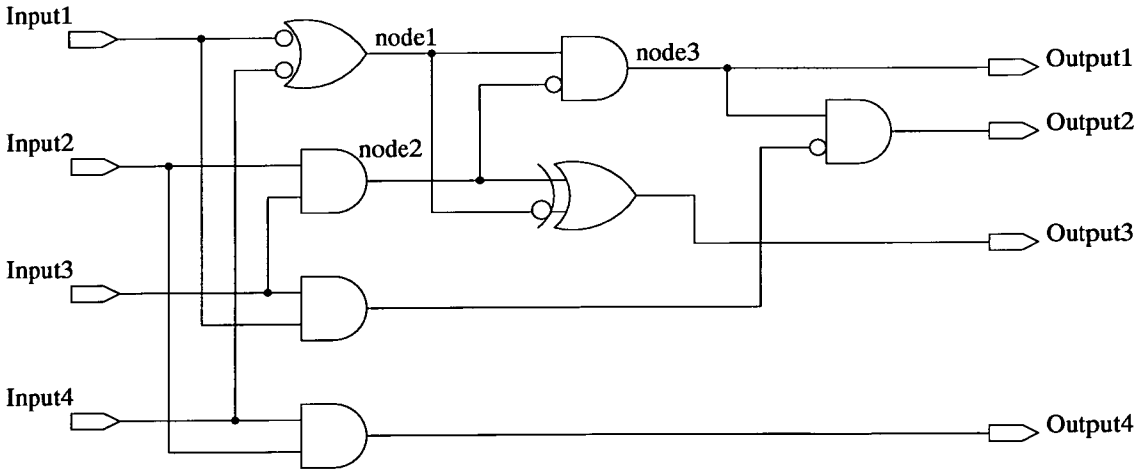


Figure A.1: *2x2-bit parallel multiplier evolved my Miller et.al.*

```
-------------------------------------------------------
-- Description of Miller 'Novel' 2x2 parallel multiplier
-- Taken from "Digital Circuit Evolution and Fitness
-- Landscapes", Proceedings of the 1999 Congress on
-- Evolutionary Computation, CEC 99
-------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY Miller2x2mult IS
```

```
  PORT(SIGNAL Input  : IN std_logic_vector(3 DOWNTO 0);
       SIGNAL Output : OUT std_logic_vector(3 DOWNTO 0));
END Miller2x2mult;

ARCHITECTURE rtl OF Miller2x2mult IS
BEGIN
  -- purpose: describes 2x2-bit multiplier at gate-level
  multiplier: process (Input)
    variable node1, node2, node3 : std_logic;
  begin
    node1 := (not Input(0)) or (not Input(3));
    node2 := Input(1) and Input(2);
    node3 := node1 and (not node2);
    Output(0) <= node3;
    Output(1) <= (not node3) and (Input(0) and Input(2));
    Output(2) <= (not node1) xor node2;
    Output(3) <= Input(1) and Input(3);
  end process multiplier;
END rtl;
```

A.5 Schematic of 3x3-bit Parallel Multiplier Evolved by Miller et.al. and Associated VHDL Code

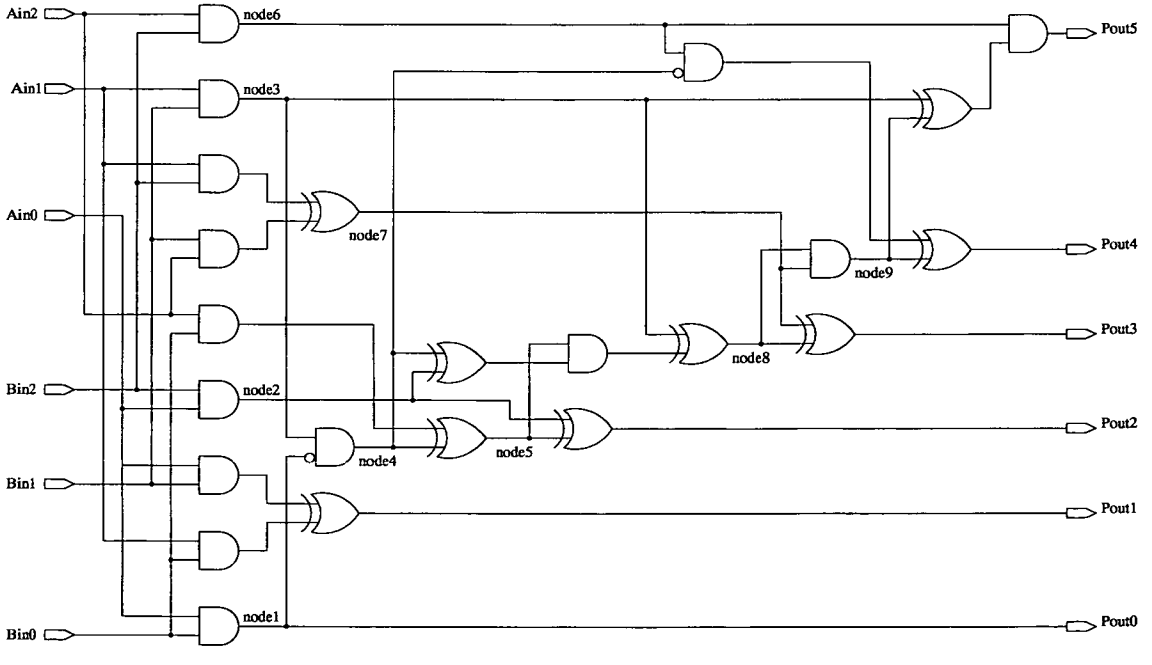


Figure A.2: *3x3-bit parallel multiplier evolved my Miller et.al.*

```
-------------------------------------------------------
-- Description of Miller 'Novel' 3x3 parallel multiplier
-- Taken from "Towards the Automatic Design of More
-- Efficient Digital Circuits", In Proceedings of the
-- Second NASA/DOD Workshop on Evolvable Hardware, 2000
-------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY Miller3x3mult IS
  PORT(SIGNAL Ain, Bin : IN std_logic_vector(2 DOWNTO 0);
       SIGNAL Pout     : OUT std_logic_vector(5 DOWNTO 0));
END Miller3x3mult;

ARCHITECTURE rtl OF Miller3x3mult IS
BEGIN
  -- purpose: describes 3x3-bit multiplier at gate-level
  multiplier: process (Ain, Bin)
    variable node1, node2, node3, node4 : std_logic;
    variable node5, node6, node7, node8, node9 : std_logic;
  begin
  node1   := Ain(0) and Bin(0);
  Pout(0) <= node1;
  Pout(1) <= (Ain(0) and Bin(1)) xor (Ain(1) and Bin(0));
  node2   := Ain(0) and Bin(2);
  node3   := Ain(1) and Bin(1);
  node4   := node3 and (not node1);
  node5   := (Ain(2) and Bin(0)) xor node4;
  Pout(2) <= node2 xor node5;
  node6   := Ain(2) and Bin(2);
  node7   := (Ain(1) and Bin(2)) xor (Ain(2) and Bin(1));
  node8   := ((node2 xor node4) and node5) xor node3;
  Pout(3) <= node7 xor node8;
  node9   := node7 and node8;
  Pout(4) <= node9 xor ((not node4) and node6);
  Pout(5) <= (node3 xor node9) and node6;
  end process multiplier;
END rtl;
```

Appendix B
Further Details of FPGA and PLA-Based EHW Platforms

B.1 Postscript Templates of FPGA Interconnect Topologies for Graphical Representation

B.1.1 Elements of Postscript That Are Common to FPGA Interconnect Templates

```
%!
.5 .5 scale

/box_size 65 def
/x_size 8 def
/y_size 8 def
/grid_spacing 130 def

/box {
    newpath
    moveto
    gsave
    box_size 0 rlineto
    0 box_size 2 div rlineto
    yshift y_size ne
    {
        gsave
        1 0 0 setrgbcolor
        grid_spacing 2 div 0 rlineto
        stroke
        grestore
    }if
    0 box_size 2 div rlineto
    box_size neg 0 rlineto
    closepath
    stroke
    grestore

    /Courrier findfont  % Get the basic font
    15 scalefont        % Scale the font to 15 points
    setfont             % Make it the current font
} def

/tap_box {
    newpath
    moveto
    gsave
    5 setlinewidth
    box_size 0 rlineto
    0 box_size rlineto
    box_size neg 0 rlineto
    closepath
    stroke
    grestore

} def

/background_box{
    moveto
    %DRAW HIGHLIGHTED BOX
    box_size 0 rlineto       % right
    0 box_size rlineto       % up
    box_size neg 0 rlineto   % left
    0 box_size neg rlineto   % down
    % SET COLOUR OF BACKGROUND BOX TO GREY
    gsave
    .75 .75 .75 setrgbcolor
    gsave
```

```
    fill
    grestore
    stroke
    grestore
    stroke
} def

/rgb_box{
    moveto
    %DRAW HIGHLIGHTED BOX
    box_size 0 rlineto       % right
    0 box_size rlineto       % up
    box_size neg 0 rlineto   % left
    0 box_size neg rlineto   % down
} def

/left_mux {
    moveto
    box_size 2 div box_size rmoveto     % moveto start (top of box)
    gsave
    0 1 0 setrgbcolor                   % Set left Mux input green
    0 6 rlineto                         % up
    box_size neg 0 rlineto              % left
    0 box_size 4 div rlineto            % up
    stroke
    grestore
    box_size neg box_size 4 div 6 add rmoveto
    box_size 3 div 0 rlineto            % right (mux symbol bottom right)
    box_size 3 div neg 20 rlineto       % up-left
    box_size 6 div neg 0 rlineto        % left
    gsave
    0 box_size 4 div rlineto            % up
    box_size 0 rlineto                  % right
    box_size 6 div 0 rlineto            % right
    0 6 rlineto                         % up
    stroke
    grestore
    box_size 6 div neg 0 rlineto        % left
    box_size 3 div neg 20 neg rlineto   % down-left (mux symbol bottom right)
    box_size 3 div 0 rlineto            % right
    gsave
    0 0 1 setrgbcolor                   % Set left Mux input blue
    0 box_size 4 div neg rlineto        % down
    box_size neg 0 rlineto              % left
    0 box_size 2 mul neg rlineto        % down

    yshift 1 eq
    {
        xshift 1 eq
        {
            10.9 12.5 lineto            % down
            12 0  rlineto
        }if
    }if

    stroke
    grestore
    box_size 3 div 0 rlineto            % right
    stroke
} def

/bottom_mux {
    moveto
    box_size 2 div 0 rmoveto
    0 box_size 2 div neg rlineto        % down
    box_size 6 div neg 0 rlineto        % left
    box_size 3 div neg 20 neg rlineto   % down-left (mux symbol bottom right)
    box_size 3 div 0 rlineto            % right
    gsave
    0 0 1 setrgbcolor                   % Set left Mux input blue
    0 box_size neg rlineto              % down
    box_size 2 mul neg 0 rlineto        % left
    stroke
    grestore
    box_size 3 div 0 rlineto            % right
    gsave
    0 1 0 setrgbcolor                   % Set right Mux input green
    0 box_size 3 div neg rlineto        % down
    stroke
    grestore
    box_size 3 div 0 rlineto            % right
    box_size 3 div neg 20 rlineto       % up-left
    box_size 6 div neg 0 rlineto        % left
    stroke

} def

/VDD {
    /Courrier findfont                  % Get the basic font
    18 scalefont                        % Scale the font to 15 points
```

```
    setfont                              % Make it the current font

    newpath
    moveto
    box_size 3 div box_size 3 div add box_size 2 div box_size 3
    div add 20 add neg rmoveto

    15 neg 0 rmoveto
    30 0 rlineto
    gsave
    0 1 0 setrgbcolor           % Set VDD text colour to green
    31 neg 15 neg rmoveto
    (VDD) show
    grestore
    stroke
} def

/ground {
    /Courrier findfont                          % Get the basic font
    18 scalefont                                % Scale the font to 15 points
    setfont                                     % Make it the current font
    moveto
    gsave
    1 0 0 setrgbcolor                           % Set pen colour red

    0 box_size 2 div rmoveto                    % Move up half of box
    box_size box_size 3 div add neg 0 rlineto   % left
    0 box_size 3 div neg rlineto                % down
    gsave
    box_size 4 div neg 0 rmoveto                % left
    box_size 2 div 0 rlineto                    % right
    stroke
    grestore

    17 neg 15 neg rmoveto
    (VDD) show

    stroke
    grestore
} def

/right_ground {
    /Courrier findfont                          % Get the basic font
    18 scalefont                                % Scale the font to 15 points
    setfont                                     % Make it the current font
    moveto
    gsave
    1 0 0 setrgbcolor                           % Set pen colour red

    box_size box_size 2 div rmoveto             % Move up half of box
    box_size 2 div 0 rlineto                        % right
    0 box_size 3 div neg rlineto                % down
    gsave
    box_size 4 div neg 0 rmoveto                % left
    box_size 2 div 0 rlineto                    % right
    stroke
    grestore

    17 neg 15 neg rmoveto
    (VDD) show

    stroke
    grestore
} def

/routing_block_vrt {
    rmoveto
    box_size 3 div 0 rlineto      % right
    0 box_size 8 div rlineto        % up
    box_size 3 div neg 0 rlineto  % left
    0 box_size 8 div neg rlineto    % down
} def

/routing_block_hrz {
    rmoveto
    box_size 8 div 0 rlineto      % right
    0 box_size 3 div rlineto      % up
    box_size 8 div neg 0 rlineto  % left
    0 box_size 3 div neg rlineto  % down
} def

/addition_block {
    rmoveto
    % DRAW BOX
    box_size 2 div 0 rlineto      % right
    0 box_size 2 div rlineto      % up
    box_size 2 div neg 0 rlineto  % left
    0 box_size 2 div neg rlineto  % down

    % SET COLOUR OF ADDITION BLOCK TO YELLOW
```

```
    gsave
    1 1 0 setrgbcolor
    gsave
    fill
    grestore
    stroke
    grestore

    % DRAW ADDITION SYMBOL
    gsave
    box_size 4 div box_size 8 div rmoveto
    0 box_size 4 div rlineto      % up
    box_size 8 div neg box_size 8 div neg rmoveto
    box_size 4 div 0 rlineto      % right
    stroke
    grestore
    stroke
}def

/subtraction_block {
    rmoveto
    % DRAW BOX
    box_size 2 div 0 rlineto      % right
    0 box_size 2 div rlineto      % up
    box_size 2 div neg 0 rlineto  % left
    0 box_size 2 div neg rlineto  % down

    % SET COLOUR OF SUBTRACTION BLOCK TO YELLOW
    gsave
    1 1 0 setrgbcolor
    gsave
    fill
    grestore
    stroke
    grestore

    % DRAW SUBTRACT SYMBOL
    gsave
    box_size 8 div box_size 4 div rmoveto
    box_size 4 div 0 rlineto      % right
    stroke
    grestore
    stroke
}def

/Shifter_block {
    /shift_value exch def

    % DRAW BOX
    box_size 2 div 0 rlineto      % right
    0 box_size 2 div rlineto      % up
    box_size 2 div neg 0 rlineto  % left
    0 box_size 2 div neg rlineto  % down

    % SET COLOUR OF SUBTRACTION BLOCK TO YELLOW
    gsave
    1 1 0 setrgbcolor
    gsave
    fill
    grestore
    stroke
    grestore

    % DRAW VALUE OF LEFT SHIFT
    gsave
    /Courrier findfont % Get the basic font
    19 scalefont       % Scale the font to 19 points
    setfont            % Make it the current font
    5 12 rmoveto
    (S) show
    1 0 rmoveto
    shift_value (xxxx) cvs show
    grestore
    stroke
}def
```

B.1.2 Postscript Template for Alternating Feed-Forward Array (AFFA) FPGA Interconnect Topology

```
%%%%%%%%%%%%% START PROGRAM %%%%%%%%%%%%%%

1 1 x_size
{
    /xshift exch def
    1 1 y_size
    {
        /yshift exch def

        % DRAW BOTTOM MUX AND BOTTOM WRAP CONNECT
        xshift 1 eq
        {
            yshift grid_spacing mul xshift grid_spacing mul bottom_mux
            yshift grid_spacing mul xshift grid_spacing mul VDD
        }if

        % DRAW THE BOX
        yshift grid_spacing mul xshift grid_spacing mul box
        yshift 1 eq
        {
            stroke
        }
        {
            %gsave
            %1 1 1 setrgbcolor
            %.86 .86 .25 setrgbcolor
            %gsave
            %fill
            %grestore
            stroke
            %grestore
        }ifelse

        % DRAW LEFT MUX
        yshift 1 eq
        { %if
            xshift x_size eq not
            { %if
                yshift grid_spacing mul xshift grid_spacing mul left_mux
            }if

            % ALTERNATE GROUND CONNECTION
            xshift 2 mod 0 ne
            {
                % Draw ground connection to PALU
                yshift grid_spacing mul xshift grid_spacing mul ground
            }if
        }
        { %else
            yshift grid_spacing mul xshift grid_spacing mul moveto
            box_size 2 div box_size rmoveto
            x_size xshift eq not
            {
                gsave
                0 1 0 setrgbcolor
                0 grid_spacing 2 div rlineto
                stroke
                grestore
            }if
        }ifelse

        % DRAW FAR RIGHT GROUND
        yshift y_size eq
        {
            xshift 2 mod 0 eq
            {
                % Draw ground connection to PALU
                yshift grid_spacing mul xshift grid_spacing mul right_ground
            }if
        }if

    } for
} for
```

B.1.3 Postscript Template for Continuous Feed-Forward Array (CFFA) FPGA Interconnect Topology

```
%%%%%%%%%%%%% START PROGRAM %%%%%%%%%%%%%%

1 1 x_size
{
    /xshift exch def
    1 1 y_size
    {
        /yshift exch def

        % DRAW BOTTOM MUX AND BOTTOM WRAP CONNECT
        xshift 1 eq
        {
            yshift grid_spacing mul xshift grid_spacing mul bottom_mux
            yshift grid_spacing mul xshift grid_spacing mul VDD
        }if

        % DRAW THE BOX
        yshift grid_spacing mul xshift grid_spacing mul box
        yshift 1 eq
        {
            stroke
        }
        {
            %gsave
            %1 1 1 setrgbcolor
            %.86 .86 .25 setrgbcolor
            %gsave
            %fill
            %grestore
            stroke
            %grestore
        }ifelse

        % DRAW LEFT MUX
        yshift 1 eq
        { %if
            xshift x_size eq not
            { %if
                yshift grid_spacing mul xshift grid_spacing mul left_mux
            }if

            % Draw ground connection to PALU
            yshift grid_spacing mul xshift grid_spacing mul ground
        }
        { %else
            yshift grid_spacing mul xshift grid_spacing mul moveto
            box_size 2 div box_size rmoveto
            x_size xshift eq not
            {
                gsave
                0 1 0 setrgbcolor
                0 grid_spacing 2 div rlineto
                stroke
                grestore
            }if
        }ifelse

    } for
} for
```

B.1.4 Postscript Template for Continuous Feed-Forward Loop Array (CLFFA) FPGA Interconnect Topology

```
/wrap {
    /Courrier findfont                  % Get the basic font
    18 scalefont                        % Scale the font to 15 points
    setfont                             % Make it the current font

    newpath
    moveto
    xshift x_size eq
    { %if
        box_size 2 div box_size 2 mul box_size 3 div add rmoveto
        15 neg 0 rmoveto
        30 0 rlineto
        0 box_size 3 div neg rlineto        % down
        30 neg 0 rlineto
        closepath
        gsave
        0 1 0 setrgbcolor
        gsave
        fill
```

```
        grestore
        stroke
        grestore
        5 16 neg rmoveto
        yshift (xxxx) cvs show
    }
    { %else
        box_size 3 div box_size 3 div add box_size 2 div box_size 3
        div add 20 add neg rmoveto

        15 neg 0 rmoveto
        30 0 rlineto
        yshift 1 eq
        { %if
            gsave
            0 1 0 setrgbcolor           % Set VDD text colour to green
            31 neg 15 neg rmoveto
            (VDD) show
            grestore
        }
        {%else
            0 box_size 3 div neg rlineto        % down
            30 neg 0 rlineto
            closepath
            gsave
            0 1 0 setrgbcolor
            gsave
            fill
            grestore
            stroke
            grestore
            5 16 neg rmoveto
            yshift 1 sub (xxxx) cvs show
        }ifelse
    }ifelse

    stroke
} def

%%%%%%%%%%%% START PROGRAM %%%%%%%%%%%%%

1 1 x_size
{
    /xshift exch def
    1 1 y_size
    {
        /yshift exch def

        % DRAW BOTTOM MUX AND BOTTOM WRAP CONNECT
        xshift 1 eq
        {
            yshift grid_spacing mul xshift grid_spacing mul bottom_mux
            yshift grid_spacing mul xshift grid_spacing mul wrap
        }if

        % DRAW THE BOX
        yshift grid_spacing mul xshift grid_spacing mul box
        yshift 1 eq
        {
            stroke
        }
        {
            %gsave
            %1 1 1 setrgbcolor
            %.86 .86 .25 setrgbcolor
            %gsave
            %fill
            %grestore
            stroke
            %grestore
        }ifelse

        % DRAW LEFT MUX
        yshift 1 eq
        { %if
            xshift x_size eq not
            { %if
                yshift grid_spacing mul xshift grid_spacing mul left_mux
            }
            { %else
                yshift grid_spacing mul xshift grid_spacing mul moveto
                box_size 2 div box_size rmoveto
                gsave
                0 1 0 setrgbcolor
                0 grid_spacing 2 div rlineto
                stroke
                grestore
            }ifelse

            % Draw ground connection to PALU
```

```
            yshift grid_spacing mul xshift grid_spacing mul ground
        }
        { %else
            yshift grid_spacing mul xshift grid_spacing mul moveto
            box_size 2 div box_size rmoveto
            y_size x_size mul yshift xshift mul eq not
            {
                gsave
                0 1 0 setrgbcolor
                0 grid_spacing 2 div rlineto
                stroke
                grestore
            }if
        }ifelse

        % DRAW TOP WRAP CONNECTS
        xshift x_size eq
        {
            yshift y_size eq not
            {
                yshift grid_spacing mul xshift grid_spacing mul wrap
            }if
        }if

    } for
} for
```

B.2 Postscript Templates of PLA Interconnect Topologies for Graphical Representation

B.2.1 Elements of Postscript That Are Common to PLA Interconnect Templates

```
%%Orientation: Landscape

/box_size 65 def
/grid_spacing_vrt box_size 2 mul def
/grid_spacing_hrz box_size 3 mul def

yscale xscale scale
90 rotate
0 grid_spacing_vrt x_size 2 add mul neg translate

/colourA{
    .7 .7 1 setrgbcolor
} def

/colourB{
    1 .4 .4 setrgbcolor
} def

/colourC{
    1 .8 0 setrgbcolor
} def

/colourD{
    .2 .7 .8 setrgbcolor
} def

/colourE{
    .5 .8 0 setrgbcolor
} def

/PALU{
    /Courier findfont               % Get the basic font
    25 scalefont                    % Scale the font to 15 points
    setfont                         % Make it the current font

    newpath
    moveto
    box_size 0 rlineto              % right
    0 box_size 2 div rlineto        % up

    yshift y_size ne
    {
        gsave
        box_size 2 div 0 rlineto        % right

        % DISPLAY OUTPUT NUMBER
        box_size 3 div neg 5 rmoveto
        xshift 1 sub  (xxxx) cvs show

        stroke
        grestore
```

```
    }if

    0 box_size 2 div rlineto        % up
    box_size neg 0 rlineto          % left
    0 box_size 4 div neg rlineto    % down
    gsave
    box_size 2 div neg 0 rlineto    % left
    stroke
    grestore
    0 box_size 2 div neg rlineto    % down
    gsave
    box_size 2 div neg 0 rlineto    % left
    stroke
    grestore
    0 box_size 4 div neg rlineto    % down
    closepath
    gsave

    % SET COLOUR OF ROUTING BLOCK FOR CONNECTION CLARITY
    yshift 2 mod 1 eq
    {
        colourA
    }
    {
        colourB
    }ifelse

    yshift X2_colour eq
    {
        colourC
        /X2_colour X2_colour 4 add store
    }if
    yshift X4_colour eq
    {
        colourD
        /X4_colour X4_colour 6 add store
    }if
    yshift X4_colour 4 sub eq
    {
        colourE
    }if
    gsave
    fill
    grestore
    stroke
    grestore

} def

/Shifter{
    /Courier findfont               % Get the basic font
    25 scalefont                    % Scale the font to 15 points
    setfont                         % Make it the current font

    newpath
    moveto
    box_size 0 rlineto              % right
    0 box_size 2 div rlineto        % up

    gsave
    box_size 2 div 0 rlineto        % right

    % DISPLAY OUTPUT NUMBER
    box_size 3 div neg 5 rmoveto
    xshift 1 sub  (xxxx) cvs show

    stroke
    grestore

    0 box_size 2 div rlineto        % up
    box_size neg 0 rlineto          % left
    0 box_size 2 div neg rlineto    % down
    gsave
    box_size 2 div neg 0 rlineto    % left
    stroke
    grestore
    0 box_size 2 div neg rlineto    % down
    closepath

    gsave
    % SET COLOUR OF Shifter
    colourA
    gsave
    fill
    grestore
    stroke
    grestore


} def
```

```
/input_bus{
    /Courier findfont               % Get the basic font
    20 scalefont                    % Scale the font to 25 points
    setfont                         % Make it the current font

    gsave
    yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul moveto
    box_size 2 div neg box_size 2 div rmoveto
    0 grid_spacing_vrt x_size 1 sub mul 2 div rlineto
    gsave
    100 neg 0 rlineto

    % SHOW IMPULSE STRING
    0 5 rmoveto
    (IMPULSE) show

    stroke
    grestore
    0 grid_spacing_vrt x_size 1 sub mul 2 div rlineto
    stroke
    grestore
} def

/routing {

    box_size 0 rmoveto                % move to bottom right of PALU at
    box_size 2 div box_size 2 div neg rmoveto   % 1/2 box_size distance

    box_size 0 rlineto                  % right
    0 box_size x_size mul rlineto       % up
    0 grid_spacing_vrt 2 div x_size mul rlineto       % up
    box_size neg 0 rlineto              % left
    0 box_size x_size mul neg rlineto   % down
    0 grid_spacing_vrt 2 div x_size mul neg rlineto   % down
    stroke
} def

/connection_block{
    rmoveto
    box_size 1.5 div 0 rlineto      % right
    0 box_size 2 div rlineto        % up
    box_size 1.5 div neg 0 rlineto  % left
    0 box_size 2 div neg rlineto    % down
} def

/addition_block{
    rmoveto
    % DRAW BOX
    box_size 2 div 0 rlineto      % right
    0 box_size 2 div rlineto      % up
    box_size 2 div neg 0 rlineto  % left
    0 box_size 2 div neg rlineto  % down

    % SET COLOUR OF ADDITION BLOCK TO WHITE
    gsave
    1 1 1 setrgbcolor
    gsave
    fill
    grestore
    stroke
    grestore

    % DRAW ADDITION SYMBOL
    gsave
    box_size 4 div box_size 8 div rmoveto
    0 box_size 4 div rlineto      % up
    box_size 8 div neg box_size 8 div neg rmoveto
    box_size 4 div 0 rlineto      % right
    stroke
    grestore
    stroke
}def

/subtraction_block{
    rmoveto
    % DRAW BOX
    box_size 2 div 0 rlineto      % right
    0 box_size 2 div rlineto      % up
    box_size 2 div neg 0 rlineto  % left
    0 box_size 2 div neg rlineto  % down

    % SET COLOUR OF SUBTRACTION BLOCK TO WHITE
    gsave
    1 1 1 setrgbcolor
    gsave
    fill
    grestore
    stroke
    grestore
```

```
    % DRAW SUBTRACT SYMBOL
    gsave
    box_size 8 div box_size 4 div rmoveto
    box_size 4 div 0 rlineto      % right
    stroke
    grestore
    stroke
}def

/Shifter_block{
    rmoveto
    % DRAW BOX
    box_size 2 div box_size 4 div add 0 rlineto      % right
    0 box_size 2 div rlineto      % up
    box_size 2 div box_size 4 div add neg 0 rlineto  % left
    0 box_size 2 div neg rlineto  % down

    % SET COLOUR OF SUBTRACTION BLOCK TO WHITE
    gsave
    1 1 1 setrgbcolor
    gsave
    fill
    grestore
    stroke
    grestore

    stroke
}def

/X2_fast_route{
    box_size 4 div 0 rmoveto
    box_size 2 mul box_size 2 div neg rmoveto

    % SET TWO ROUTE LINES SO CONNECTIONS ARE MORE VISABLE
    yshift 2 mod 0 eq
    {
        0 box_size 2 div neg rlineto      % down
        grid_spacing_hrz 2 mul box_size 2 div sub 0 rlineto  % right
        0 box_size 2 div rlineto          % up
    }
    {
        0 box_size neg rlineto      % down
        grid_spacing_hrz 2 mul box_size 2 div sub 0 rlineto  % right
        0 box_size rlineto          % up
    }ifelse
    stroke
}def

/X4_fast_route{
    box_size 4 div 0 rmoveto
    box_size 2 mul box_size 2 div neg rmoveto

    % SET TWO ROUTE LINES SO CONNECTIONS ARE MORE VISABLE
    yshift 4 mod 0 eq
    {
        0 box_size 1.5 mul neg rlineto                      % down
        grid_spacing_hrz 4 mul box_size 2 div sub 0 rlineto % right
        0 box_size 1.5 mul rlineto                          % up
    }
    {
        0 box_size neg rlineto                              % down
        grid_spacing_hrz 4 mul box_size 2 div sub 0 rlineto % right
        0 box_size rlineto                                  % up
    }ifelse

    stroke
}def

/highlight_box{
    rmoveto
    %DRAW HIGHLIGHTED BOX
    box_size 0 rlineto       % right
    0 box_size rlineto       % up
    box_size neg 0 rlineto   % left
    0 box_size neg rlineto   % down
    closepath
    7 setlinewidth
    0 0 0 setrgbcolor
    stroke
} def

/background_box{
    rmoveto
    %DRAW HIGHLIGHTED BOX
    box_size 0 rlineto       % right
    0 box_size rlineto       % up
    box_size neg 0 rlineto   % left
    0 box_size neg rlineto   % down
    % SET COLOUR OF BACKGROUND BOX TO GREY
```

```
    gsave
    .75 .75 .75 setrgbcolor
    gsave
    fill
    grestore
    stroke
    grestore
    stroke
} def
```

B.2.2 Postscript Template for *Route 1* PLA Interconnect Topology

```
%%%%%%%%%%%%%%%% GENERATE ROUTE 1 PLA TEMPLATE %%%%%%%%%%%%%%%%
1 1 x_size
{
    /X2_colour 3 def
    /X4_colour 2 def
    /xshift exch def

    1 1 y_size
    {
        /yshift exch def


        % DRAW SHIFTER
        yshift 1 eq
        {
            yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul Shifter

            % DRAW INPUT BUS
            xshift 1 eq
            {
                input_bus
            }if
        }
        % DRAW PALU
        {
            yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul PALU
        }ifelse

        yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul moveto

        % DRAW INTERCONNECT BOX

        xshift 1 eq
        {
            yshift y_size ne
            {
                % DRAW ROUTING BLOCK
                gsave
                routing
                grestore
            }if
        }if
        stroke

    } for

} for
```

B.2.3 Postscript Template for *Route 2* PLA Interconnect Topology

```
%%%%%%%%%%%%%%%% GENERATE ROUTE 2 PLA TEMPLATE %%%%%%%%%%%%%%%%
1 1 x_size
{
    /X2_colour 1 def
    /xshift exch def

    1 1 y_size
    {
        /yshift exch def


        % DRAW SHIFTER
        yshift 1 eq
        {
            yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul Shifter

            % DRAW INPUT BUS
            xshift 1 eq
            {
                input_bus
            }if
        }
        % DRAW PALU
```

```
        {
            yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul PALU
        }ifelse

        yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul moveto

        % DRAW INTERCONNECT BOX

        xshift 1 eq
        {
            yshift y_size ne
            {
                % DRAW ROUTING BLOCK
                gsave
                routing
                grestore
                stroke

                % DRAW 2X FAST INTERCONNECT
                yshift y_size 2 sub lt
                {
                    gsave
                    yshift grid_spacing_hrz mul xshift grid_spacing_vrt
                    mul X2_fast_route
                    grestore
                }if
            }if
        }if
        stroke

    } for

} for
```

B.2.4 Postscript Template for *Route 3* PLA Interconnect Topology

```
%%%%%%%%%%%%%%%% GENERATE ROUTE 3 PLA TEMPLATE %%%%%%%%%%%%%%%%
1 1 x_size
{
    /X2_colour 3 def
    /X4_colour 2 def
    /xshift exch def

    1 1 y_size
    {
        /yshift exch def


        % DRAW SHIFTER
        yshift 1 eq
        {
            yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul Shifter

            % DRAW INPUT BUS
            xshift 1 eq
            {
                input_bus
            }if
        }
        % DRAW PALU
        {
            yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul PALU
        }ifelse

        yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul moveto

        % DRAW INTERCONNECT BOX

        xshift 1 eq
        {
            yshift y_size ne
            {
                % DRAW ROUTING BLOCK
                gsave
                routing
                grestore

                % DRAW 2X FAST INTERCONNECT ON ODD 'YSHIFTS' ONLY
                yshift 2 mod 1 eq
                {
                    yshift y_size 2 sub lt
                    {
                        gsave
                        X2_fast_route
                        grestore
                    }if
                }if
                % DRAW 4X FAST INTERCONNECT ON EVEN 'YSHIFTS' ONLY
```

```
                yshift 2 mod 1 ne
                {
                    yshift y_size 4 sub lt
                    {
                        gsave
                        X4_fast_route
                        grestore
                    }if
                }if
            }if
        }if
        stroke

    } for

} for
```

B.2.5 Postscript Template for *Route 4* PLA Interconnect Topology

```
%%%%%%%%%%%%%%%% GENERATE ROUTE 4 PLA TEMPLATE %%%%%%%%%%%%%%%%
1 1 x_size
{
    /X2_colour 3 def
    /X4_colour 2 def
    /xshift exch def

    1 1 y_size
    {
        /yshift exch def


        % DRAW SHIFTER
        yshift 1 eq
        {
            yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul Shifter

            % DRAW INPUT BUS
            xshift 1 eq
            {
                input_bus
            }if
        }
        % DRAW PALU
        {
            yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul PALU
        }ifelse

        yshift grid_spacing_hrz mul xshift grid_spacing_vrt mul moveto

        % DRAW INTERCONNECT BOX

        xshift 1 eq
        {
            yshift y_size ne
            {
                % DRAW ROUTING BLOCK
                gsave
                routing
                grestore

                % DRAW 2X FAST INTERCONNECT ON ODD 'YSHIFTS' ONLY
                yshift 2 mod 1 eq
                {
                    yshift y_size 2 sub lt
                    {
                        gsave
                        X2_fast_route
                        grestore
                    }if
                }if
                % DRAW 4X FAST INTERCONNECT ON EVEN 'YSHIFTS' ONLY
                yshift 2 mod 1 ne
                {
                    yshift y_size 4 sub lt
                    {
                        gsave
                        X4_fast_route
                        grestore
                    }if
                }if
            }if
        }if
        stroke

    } for

} for
```

Appendix C

Synthesis and Simulation Script for Generation of 6x5 PLA Core

C.1 Top-Down Synthesis script for 6x5 PLA Core

```
LOG_FILE = "/tmp/EHW_platform/synopsis/log_files/ \
TD_100MHz_6X5_PLA_C2_limited_V3.log"
MASU_FILE ="TD_100MHz_6X5_PLA_C2"
NETDIR = "/tmp/EHW_platform/synopsis/reports/"
PLOTS = "/tmp/EHW_platform/synopsis/plots/"

/* Parameters for Clock, Reset, In- and Outputs: */

   CLKNAME = "clock"
   RESNAME = "GlobalReset"
   CLKPERIOD = 10.0
   CLKHP = CLKPERIOD / 2
   CLKSKEW = 1
   CLKTRANS = 0.5
   CLKDELAY = 1
   INPDELAY = 1
   OUTPDELAY = 1

/* Parameters for PLA Elaboration: */

   BUSWIDTH = 16        /* I/O Bus width of EHW environment */
   XWIDTH = 6           /* Number of PALUs in X Axis */
   YWIDTH = 5           /* Number of PALUs in Y Axis */
   CONNECT_CNTRL = 3    /* Bit width of control for interconnect_mux */
   PALU_CNTRL = 5       /* Bit width of control for each PALU */
   CIRCUITOUTPUTS = 5   /* Number of circuit outputs required */
   MAX_CONNECT = 3      /* Number of PALUs connected to Interconnect_Mux*/
   CNTRL_OFFSET = 20    /* Number of bits required to control MaxShifters*/

MASU = "PLA_C2_limited_V3"
VER = "_limited_V3"
MASU_VER = MASU_FILE + VER

analyze -f vhdl -lib WORK VHDSRC + pack_local.vhd
analyze -f vhdl -lib WORK VHDSRC + MUX_2_FFA_cells.vhd
analyze -f vhdl -lib WORK VHDSRC + addsub_cla.vhd
analyze -f vhdl -lib WORK VHDSRC + NbitMux_2in.vhd
analyze -f vhdl -lib WORK VHDSRC + Npos_leftshift.vhd
analyze -f vhdl -lib WORK VHDSRC + Maxshift_limited16.vhd
analyze -f vhdl -lib WORK VHDSRC + POF_ALU.vhd
analyze -f vhdl -lib WORK VHDSRC + Nbit_shiftEnable.vhd
analyze -f vhdl -lib WORK VHDSRC + Nbit_ShiftReg.vhd
analyze -f vhdl -lib WORK VHDSRC + Interconnect_Mux_limited.vhd
analyze -f vhdl -lib WORK VHDSRC + Interconnect_Mux2_limited.vhd
analyze -f vhdl -lib WORK VHDSRC + Nbit_ser_to_par.vhd
analyze -f vhdl -lib WORK VHDSRC + PLA_C2_limited_V3.vhd

include SCRIPT + "PLA_parameters" + VER + ".scr"

sh date

elaborate MASU -param "BusWidth=" + BUSWIDTH + ",Xwidth=" + XWIDTH + \
",Ywidth=" + YWIDTH + ",Connect_Cntrl=" + CONNECT_CNTRL + ",PALU_Cntrl=" +\
 PALU_CNTRL + ",CircuitOutputs=" + CIRCUITOUTPUTS + ",Max_Connect=" + \
MAX_CONNECT > NETDIR + MASU_VER + "_elaboration.rpt"

current_design = MASU
set_operating_conditions -min BCIND -max WCIND -lib MTC45000.db:MTC45000
set_wire_load_model -name 36000to42000 -lib \
MTC45000_WL_WORST.db:MTC45000_WL_WORST -max

set_wire_load_model -name 36000to42000 -lib \
MTC45000_WL_TYP.db:MTC45000_WL_TYP -min

create_clock CLKNAME -period CLKPERIOD -waveform {0 CLKHP}
set_clock_uncertainty CLKSKEW CLKNAME
```

```
set_clock_transition CLKTRANS CLKNAME
set_dont_touch_network {CLKNAME RESNAME}
set_drive 0 {CLKNAME,RESNAME}
set_input_delay INPDELAY -add_delay -clock CLKNAME all_inputs() >> LOG_FILE
set_output_delay OUTPDELAY -add_delay -clock CLKNAME all_outputs() >> LOG_FILE
set_fix_hold CLKNAME

uniquify

current_design MASU
compile -map_effort medium >> LOG_FILE
change_names -h -rules NET >> LOG_FILE
report_constraint -all_violators > NETDIR + MASU_VER + "_violations.rpt"

report_area >> NETDIR + MASU_VER + ".rpt"
report_timing -delay max >> NETDIR + MASU_VER + ".rpt"

write -f vhdl -h -output NETDIR + MASU_VER + ".vhd"
write -f verilog -h -output NETDIR + MASU_VER + ".v"
write -f db -h -output NETDIR + MASU_VER + ".db"
remove_design -all >> LOG_FILE

exit
```

C.2 VHDL Leapfrog Testbench for Netlist Simulation 6x5 PLA Core

```
-------------------------------------------------------------------
-- PLA architecture for development of FIR Filter algorithms
-------------------------------------------------------------------
LIBRARY ieee;
library mtc_lib;

USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE WORK.local.ALL;
use mtc_lib.MTC45000_Vcomponents.all;
use work.CONV_PACK_PLA_C2_limited_V3.all;

ENTITY PLA_C2_limited_V3_testbench IS

END PLA_C2_limited_V3_testbench;

ARCHITECTURE functional OF PLA_C2_limited_V3_testbench IS

  ---------------------------------------------------------------------------------------------
  -- Constants are user defined and determine dimensions of PLA architecture
  ---------------------------------------------------------------------------------------------
  CONSTANT BusWidth         : integer := 16;    -- I/O Bus width of EHW environment
  CONSTANT Xwidth           : integer := 6;    -- Number of EHW CLBs in X Axis
  CONSTANT Ywidth           : integer := 5;     -- Number of EHW CLBs in Y Axis
  CONSTANT CircuitOutputs   : integer := 5;     -- Number of circuit outputs required
  CONSTANT PALU_Cntrl       : integer := 5;     -- Bit width of control for
                                                -- each Programmable ALU
  CONSTANT Max_Connect      : integer := 3;     -- Number of PALUs
                                                -- connected to Routing logic
  constant Connect_Cntrl    : integer := log_2((2*Max_Connect)-1)+1;  -- Bit width of
                                                                      -- control for
                                                                      -- routing MUXs

  constant Cntrl_offset : integer := ((log_2(BusWidth-1)+1) * Ywidth);  --   Control
                                                                        --   offset
                                                                        --   for
                                                                        --   leftmost
                                                                        --   shifters
  constant string_length : integer := (((Xwidth-1) * Ywidth *
                                        (PALU_Cntrl+(2*Connect_Cntrl))) +
                                       Cntrl_offset);

  constant initial_delay : integer := 10;         -- Number of clock cycles before
                                                  -- loading configuration data

  type Output_pins is array (1 to CircuitOutputs) of std_logic_vector(BusWidth-1 downto 0);
  ---------------------------------------------------------------------------------------------
  -- Global Inputs
  ---------------------------------------------------------------------------------------------
  SIGNAL clock        : std_logic;
  SIGNAL GlobalReset  : std_logic;

  ---------------------------------------------------------------------------------------------
  -- Inputs to PLA Architecture
  ---------------------------------------------------------------------------------------------
  SIGNAL PLA_SignalInput    : typeId_0;
  SIGNAL PLA_data_stream    : std_logic;
  SIGNAL Data_enable        : std_logic := '0';
```

```
SIGNAL Load_PLA           : std_logic := '0';

-------------------------------------------------------------------------------------------
-- Outputs from PLA Architecture
-------------------------------------------------------------------------------------------
SIGNAL PLA_Output_Bus : typeId_1;
SIGNAL Output_Port    : Output_pins;

-------------------------------------------------------------------------------------------
-- Inputs to Nbit_par_to_ser (temporary memory unit)
-------------------------------------------------------------------------------------------
SIGNAL load_memory     : std_logic := '1';
SIGNAL memory_contents : std_logic_vector(string_length - 1 DOWNTO 0)
  := "1101010111001001110110101010111011000010001011011001001110001001010010000001001111001
      1011010000100011110101000011100000101010110010110101011010111001000011001010001111100
      0111000110000101101011110110101001011011000110110000101110100111101000100100101010000
      1111001000111001110100001101010011001001";

component PLA_C2_limited_V3
  port(EnvironmentInput : in typeId_0;
       ChromosomeString, clock, GlobalReset, load_PLA, Enable_data : in std_logic;
       PLA_Output_Bus : out typeId_1);
end component;

COMPONENT Nbit_par_to_ser
  GENERIC(Xwidth           : integer;
          Ywidth           : integer;
          Cntrl_offset     : integer;
          PALU_Cntrl       : integer;
          Connect_Cntrl    : integer);
  PORT(SIGNAL clock        : IN std_logic;
       SIGNAL load_enable  : IN std_logic;
       SIGNAL Par_input    : IN std_logic_vector(string_length - 1 DOWNTO 0);
       SIGNAL Ser_output   : OUT std_logic);
end COMPONENT;

BEGIN
-------------------------------------------------------------------------------
-- Create individual output buses from PLA_Output_Bus
-------------------------------------------------------------------------------
OutputBus: FOR i in 1 to CircuitOutputs GENERATE
  Output_Port(i) <= (PLA_Output_Bus( (i*BusWidth)-1) &
                     PLA_Output_Bus( (i*BusWidth)-2) &
                     PLA_Output_Bus( (i*BusWidth)-3) &
                     PLA_Output_Bus( (i*BusWidth)-4) &
                     PLA_Output_Bus( (i*BusWidth)-5) &
                     PLA_Output_Bus( (i*BusWidth)-6) &
                     PLA_Output_Bus( (i*BusWidth)-7) &
                     PLA_Output_Bus( (i*BusWidth)-8) &
                     PLA_Output_Bus( (i*BusWidth)-9) &
                     PLA_Output_Bus( (i*BusWidth)-10) &
                     PLA_Output_Bus( (i*BusWidth)-11) &
                     PLA_Output_Bus( (i*BusWidth)-12) &
                     PLA_Output_Bus( (i*BusWidth)-13) &
                     PLA_Output_Bus( (i*BusWidth)-14) &
                     PLA_Output_Bus( (i*BusWidth)-15) &
                     PLA_Output_Bus( (i*BusWidth)-16));

end generate OutputBus;

-------------------------------------------------------------
-- Instantiate PLA_C2_limited_V3
-------------------------------------------------------------

PLA_Unit: PLA_C2_limited_V3
PORT MAP(PLA_SignalInput,
         PLA_data_stream,
         clock,
         GlobalReset,
         Load_PLA,
         Data_enable,
         PLA_Output_Bus);

-------------------------------------------------------------
-- Instantiate temporary memory unit Nbit_par_to_ser
-------------------------------------------------------------
MEM_register: Nbit_par_to_ser
  GENERIC MAP(Xwidth,
              Ywidth,
              Cntrl_offset,
              PALU_Cntrl,
              Connect_Cntrl)
  PORT MAP(clock,
           load_memory,
           memory_contents,
           PLA_data_stream);

-------------------------------------------------------------------------------
-- Counter for loading of PLA configuration string
-------------------------------------------------------------------------------
```

```
  counter: process(clock, GlobalReset)
    type input_strings is array (1 to 10) of typeId_0;
    variable count      : integer := 0;
    variable ip_count   : integer := 0;
    variable input_data : input_strings := ( 1 =>  "0000000000000001", -- 1
                                             2 =>  "0000000000011001", -- 25
                                             3 =>  "0000000000110001", -- 49
                                             4 =>  "0000000110000001", -- 385
                                             5 =>  "0000001000101001", -- 553
                                             6 =>  "0000000100001111", -- 271
                                             7 =>  "0000000001010101", -- 85
                                             8 =>  "0000000000000001", -- 1
                                             9 =>  "0000000111000001", -- 449
                                             10 =>  "0000000001101011"); -- 107
  begin
    if GlobalReset = '1' then
      Load_PLA <= '0';
      load_memory <= '1';
      Data_enable <= '0';
      count := 0;
      ip_count := 1;
    elsif CLOCK'EVENT and clock = '1' then
      count := count + 1;
      if count < (string_length + initial_delay) then
        if count = initial_delay then
          Load_PLA <= '1';
          Data_enable <= '1';
          load_memory <= '0';
          PLA_SignalInput <= "0000000000000001"; -- after 0 ns;
        end if;

      else
        Load_PLA <= '0';
        if (count = ((ip_count * initial_delay) + string_length)) then
          ip_count := ip_count + 1;
          if(ip_count < 12) then
            PLA_SignalInput <= input_data(ip_count-1);
          end if;
        end if;
      end if;
    end if;
  end process counter;
  ------------------------------------------------------------------------------
  -- Set Input Vectors for testbench analyses of PLA Architecture
  ------------------------------------------------------------------------------
  GenerateClock: PROCESS
  BEGIN
    clock <= '1' AFTER 0 ns;
    FOR i IN 0 TO 500 LOOP
      clock <= '0' AFTER 100 ns, '1' AFTER 200 ns;
      WAIT FOR 200 ns;
    END LOOP;
    WAIT;
  END PROCESS GenerateClock;

  ResetControl: PROCESS
  BEGIN
    GlobalReset <=
      '1' AFTER 0 ns,
      '0' AFTER 247 ns;
    WAIT;
  END PROCESS ResetControl;
  ------------------------------------------------------------------------------
END functional;
```

Appendix D

Publications

D.1 Refereed Journals

B. I. Hounsell, T. Arslan, *A programmable multiplierless digital filter array for embedded SoC applications*, in IEE Electronics Letters, Vol. 37(12), pp 735-737, June 2001.

B. I. Hounsell, T. Arslan, *An embedded programmable logic array for online adaptation of multiplierless FIR filters*, Submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

D.2 Refereed Conferences

B. I. Hounsell, T. Arslan, *An Embedded programmable core for the implementation off high Performance digital filters*, Proceedings of 14th Annual IEEE International ASIC/SoC Conference, Sept. 12-14, 2001. Washington USA.

B. I. Hounsell, T. Arslan, *A novel genetic algorithm for the automated design of performance driven digital circuits*, Proceedings of IEEE Congress on Evolutionary Computation (CEC), Vol. 1, pp 601-608, July 16-19, 2000, La Hoya USA.

B. I. Hounsell, T. Arslan, *A novel evolvable hardware framework for the evolution of high performance digital circuits*, Proceedings of GECCO 2000 Vol. 1, pp 525-532, July 8-12, 2000, Las Vegas USA

D.3 Refereed Workshops

B. I. Hounsell, T. Arslan, *Evolutionary design and adaptation of digital filters within an embedded fault tolerant hardware platform*, Proceedings of 3rd NASA/DoD IEEE workshop on Evolvable Hardware, Vol. 1, pp 127-135, July 12-14, 2001, Los Angeles USA.